NAVAL POSTGRADUATE SCHOOL

Monterey, California

THESIS

A GRAPH THEORETIC ALGORITHM FOR CONTOUR
SURFACE DISPLAY GENERATION

by

Mustafa SAHINTEPE

June 1985

Thesis Advisor: Michael J. Zyda

Approved for public release; distribution is unlimited

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER	2. GOVT ACCESSION NO. AD-A159822	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) A Graph Theoretic Algorithm for Contour Surface Display Generation		5. TYPE OF REPORT & PERIOD COVERED Master's Thesis June 1985
		6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Mustafa SAHINTEPE		8. CONTRACT OR GRANT NUMBER(s)
9. PERFORMING ORGANIZATION NAME AND ADDRESS Naval Postgraduate School Monterey, CA 93943-5100		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS Naval Postgraduate School Monterey, CA 93943-5100		12. REPORT DATE June 1985
		13. NUMBER OF PAGES 153
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office)		15. SECURITY CLASS. (of this report)
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution is unlimited

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19 KEY WORDS (Continue on reverse side if necessary and identify by block number)

Contouring, contour, surface display generation

20 ABSTRACT (Continue on reverse side if necessary and identify by block number)

In this study, we develop a graphy theoretic algorithm for contour surface display generation.
The inadequacies of the currently published algorithms, with respect to contour line generation for a subgrid, are pointed out in a brief review of the available lite-ature. The algorithm developed in this study, called the Large Contourting Tree Algorithm, gets rid of the cited inadequacies.

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

S N 0102-LF-014-6601

 SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

ABSTRACT (Continued)

The core component of the introduced algorithm is a two-dimensional contouring algorithm that operates on two-dimensional slices of a larger three-dimensional grid. We present the Large Contouring Tree Algorithm in the Pascal programming language in Appendix A.

A Graph Theoretic Algorithm For Contour Surface Display Generation

by

Mustafa SAHINTEPE
Lieutenant, Turkish Air Force
B.S., Air War Academy, 1981

Submitted in partial fulfiliment of the requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
June 1985

Author: ______________________
Mustafa SAHINTEPE

Approved by: ______________________
Michael J. Zyda, Thesis Advisor

____
Daniel L. Davis, Second Reader

____
Bruce J. MacLennan, Chairman,
Department of Computer Science

____
Kneale T. Marshall,
Dean of Information and Policy Sciences

ABSTRACT

This thesis
~~In this study, we~~ develops a graph theoretic algorithm for contour surface display generation.

The inadequacies of the currently published algorithms, with respect to contour line generation for a subgrid, are pointed out in a brief review of the available literature. The algorithm developed in this study, called the Large Contouring Tree Algorithm, gets rid of the cited inadequacies.

The core component of the introduced algorithm is a two-dimensional contouring algorithm that operates on two-dimensional slices of a larger three-dimensional grid. We present the Large Contouring Tree Algorithm in the Pascal programming language is presented in Appendix A. Keywords: Theses, Computer programs; Pascal.

TABLE OF CONTENTS

I. INTRODUCTION

The purpose of this study is to create and implement a graph theoretic algorithm useful for the generation of a contour surface display. A **Large Contouring Tree Algorithm** for the operations used to generate the contour lines for a regularly subdivided grid is developed. The new algorithm solves the picture efficiency problems found in the literature [Refs. 1-8].

A. SOME DEFINITIONS

A contour defined surface display is a visual representation of a surface, either wholly or partially, by the collection of lines formed when that surface is intersected by a set of parallel planes. The lines formed on each of those planes are called **Contours**. A contour represents the set of points that belong to both the surface and the particular intersecting plane.

A formal definition is that a **Contour Surface Display** is a visual display that represents all points in a particular region of a three-space $<x,y,z>$ which satisfy the relation $f(<x,y,z>)=k$, where **k** is a constant known as the **contour level**. The visual display created by this algorithm is the collection of lines that belong to the intersection of both the set of points that satisfy the relation $f(<x,y,z>)$, and a set of regularly spaced parallel planes that pass through the region of three-space for which the relation is defined.

For this study, the function f is approximated by a discrete, three-dimensional grid created by sampling that function over the volume of interest. The three-dimensional grid contains a value at each of its defined points that corresponds to the physical quantity obtained from the function, i.e. the value associated with point (x_0,y_0,z_0) is v_0, where $f(x_0,y_0,z_0)=v_0$. In order to minimize confusion, we will specify the value at a particular grid point (x,y,z) by a(x,y,z), and will specify the value at a particular point (x,y,z) of the function by f(x,y,z).

The visual display of the contour surface is created from this three-dimensional grid by taking two-dimensional slices of the grid, and constructing the two-dimensional, planar contours for each slice at the designated contour level. A slice of a three-dimensional grid is a planar, orthogonal, two-dimensional grid assigned a constant coordinate in three-space, i.e. an x-y slice of a(<x,y,z>) corresponds notationally to a(<x,y>) for a particular z coordinate. The two-dimensional, planar contours created are the lines that satisfy the relation a(<x,y,z>)=k for a particular planar coordinate, either x,y, or z, where again k is the constant contour level. If we contour all x-y slices of three-dimensional grid at contour level k, we will have a stack of parallel contours approximating the contour surface, each planar set of contours corresponding to a particular z coordinate. If we contour all x-z slices of the three dimensional grid, we again will have a stack of parallel contours approximating the contour surface, each planar set of contours corresponding to a particular y coordinate.Likewise, if we contour all y-z slices of the three-dimensional grid, we will have a stack of parallel contours approximating the contour surface, each planar set of contours corresponding to a particular x coordinate.The assemblage of the three sets of parallel, planar contours, i.e. the simultaneous display of all the contours created for the x-y, x-z, and y-z planes of the three-dimensional grid, produces a *Chicken-Wire-Like* contour surface display (see *Figure* 1.1). The three-dimensional contour surface display described in this study is created by such a procedure.

Given that the core of the contour surface display generation algorithm is the two-dimensional slice of the three-dimensional grid, it is best that we start our study with an understanding of the operations performed on that slice. *Figure* 1.2 shows a single, x-y, two-dimensional grid, with the contours drawn corresponding to contour level 50. *Figure* 1.3 shows that same two-dimensional grid, with the contours drawn corresponding to contour level 100. The two-dimensional grid of those figures is 4x5 grid; it has four values in the x direction, and five values in the y direction. The goal of the two-dimensional contouring operation for such a grid is the determination of where lines are drawn on that grid given a fixed contour level k. In order to develop an intuitive feel for that determination mechanism, we restrict our focus to a smallest portion of the complete two-dimensional grid, the 2x2 subgrid. The 2x2 subgrid is defined to be that portion of the two-dimensional grid bounded by four adjacent grid points. In

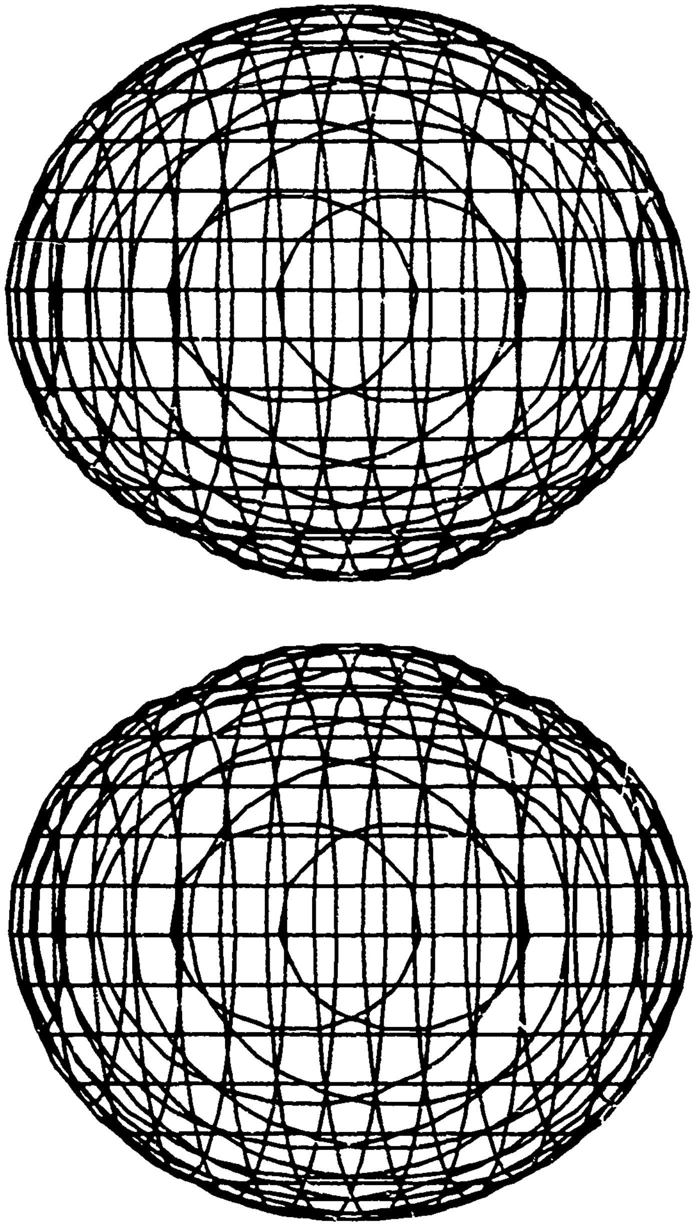


Figure 1.1
Contour Surface Display Generated from a Hydrogen Atom Wavefunction Squared (3dxy orbital)

the two-dimensional grid of *Figure* 1.2 and 1.3. the lower, lefthand 2x2 subgrid is bounded by points (1 1), (2,1), (2,2), and (1,2). The upper righthand 2x2 subgrid of the same example is bounded by points (3,4), (4,4), (4,5) and (3,5). This core algorithm has been presented in [Ref. 1] and is called the 2x2 Subgrid Algorithm.

B. A MODEL FOR CONTOURING THE 2X2 SUBGRID

The procedure used to generate the contours for a single 2x2 subgrid is the core part of two-dimensional contouring. If we compute the contours corresponding to contour level k for all 2x2 subgrids of a two-dimensional grid, then we will have determined the complete set of contours for that grid. Note that this does not make any statement as to the efficiency of that picture, i.e. there can be duplicate copies of contours, particularly for contours drawn along the border of a 2x2 subgrid. We briefly summarize the operations that comprise that procedure in order to highlight potential problems.

1. Contouring The 2x2 Subgrid

The procedure used to generate the contours for a particular 2x2 subgrid first determines if any contours should be generated for that subgrid. That determination is based upon whether any of the subgrid's edges contain the desired contour level k. An edge contains contour level k if the value of that contour level is within the range of values defined by the grid points that comprise the edge.

The next part of the contour generation procedure for the 2x2 subgrid is the computation of the contour edge intersections for any subgrid edges shown to contain the contour level. The point of intersection is computed through linear interpolation, using the grid values assigned to the endpoints of the edge and their corresponding coordinates. The point of intersection represents the location on the subgrid edge corresponding to the contour level k.

The determination of the connectivity necessary to form the appropriate contours from the list of edge intersections is the next part of the contour generation procedure. Before attempting to describe the procedure that assigns those connectivities, we first examine the subgrid's contour crossing possibilities. We accomplish that by looking at *Figure* 1.4. which shows all possible ways for contours to cross or intersect a 2x2 subgrid.

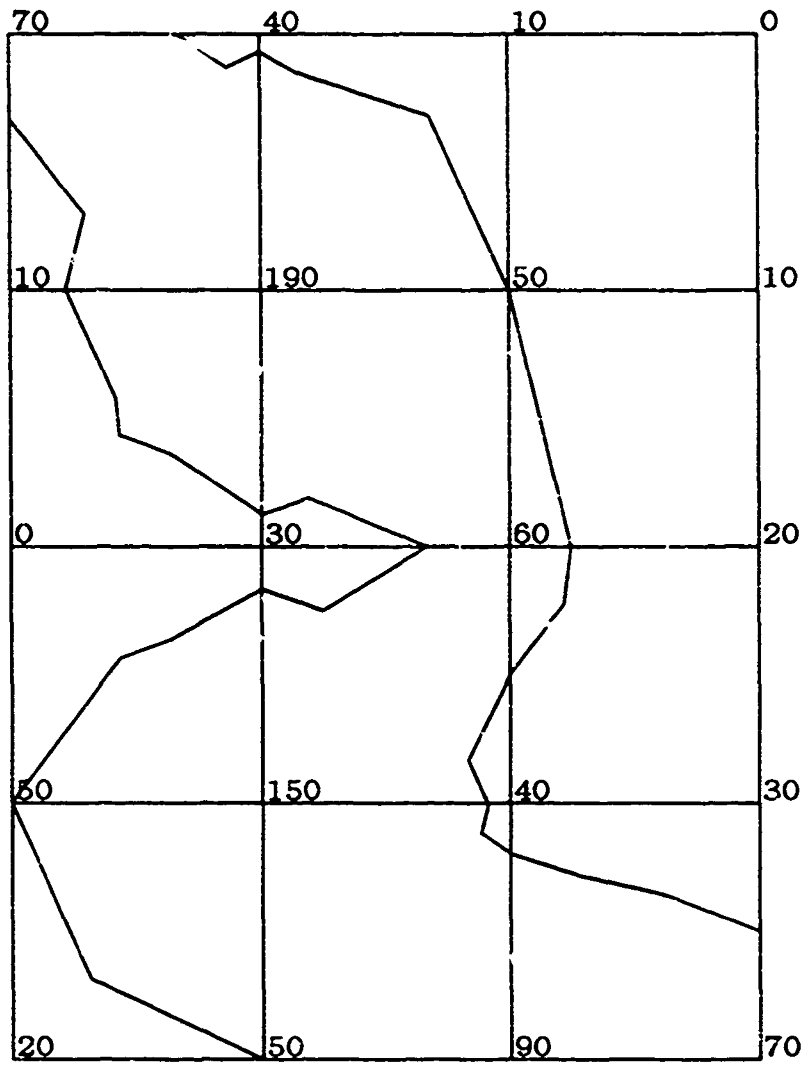


Figure 1.2
Example Contour Grid with Contours Drawn for Level 50

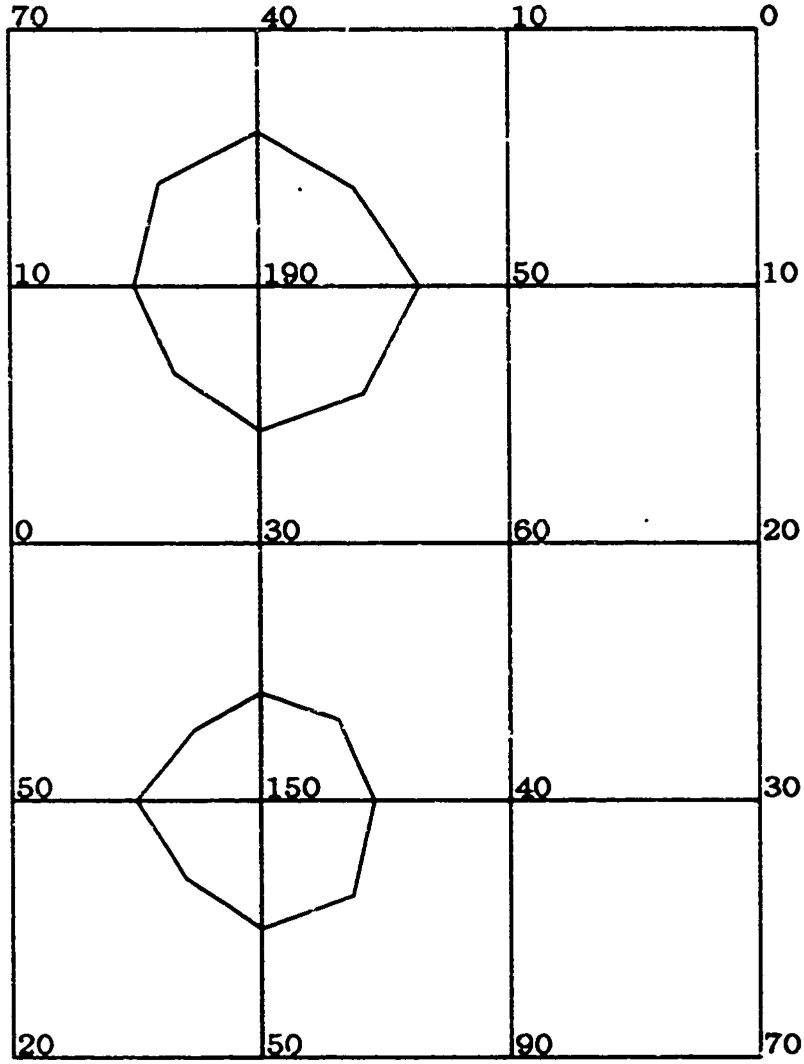


Figure 1.3
Example Contour Grid with Contours Drawn for Level 100

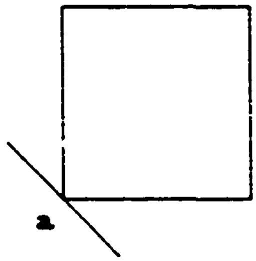
In *Figure* 1.4, there are ten cases, each of which belongs to one of three contour crossings categories:(1) single edge crossing of the 2x2, (2) double edge crossing of the 2x2, and (3) constant edge borders at the contour level for the 2x2. The ten cases are drawn according to the following small set of rules for contour crossings. (a) Contours are directed by the values associated with the edges, and are directed towards edge intersections. (b) For non-equivalued edges, if contours are indicated for a particular 2x2 subgrid, i.e. there are edges in the subgrid that contain the contour level, there is only one point of intersection for each edge of that subgrid. (c) Contours are continuous, i.e. if a contour enters a 2x2 subgrid, it must also leave that 2x2 subgrid. (d) Equivalued subgrid edges at the contour level are special cases, and are drawn in their entirety. The only exception to this rule is that constant valued 2x2 subgrids are not drawn. This is by convention.

Once we have an idea of the types of contour crossings possible for a 2x2 subgrid, and once we have an outline of the rules used in composing those possibilities, we can then address the problem of forming a procedure for assigning connectivities to the computed edge intersections. Starting with the simplest cases of *Figure* 1.4, the equivalued edge cases, we clearly see that the connectivity generation procedure for subgrids containing such edges at the contour level is relatively simple once those equivalued edges have been detected. If we find that we have a *Constant 2x2*, we do not need to issue any coordinates or connectivities because by convention we have decided not to draw that case. The other two possibilities, the *Contour Along One Edge*, or the *Contour Along Two Edges* cases, are equally as simple. The only operation necessary once such cases have been detected is to issue coordinates and connectivities corresponding to the detected edges.

At first glance, given the edge intersections for a 2x2 subgrid, the connectivity generation procedure for the single contour cases of *Figure* 1.4 seems quite easy. It appears as if the only operation that has to be done is to issue coordinates and connectivities corresponding to the straight line between the two points of edge intersection. Such a procedure works well if we know that we have a single contour crossing the subgrid. The only single contour crossing case for

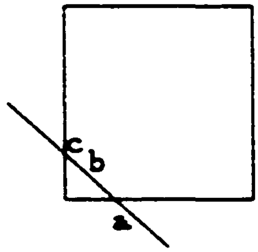
Single Contour Crossings

Case 1: Contour Tangent
to the 2 x 2



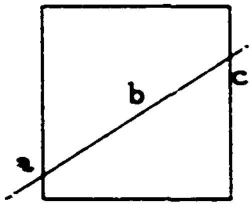
Expected Picture:
Drawpoint a

Case 2: One Contour
Through Adjacent Edges



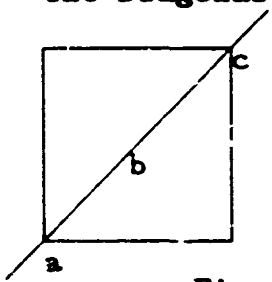
Expected Picture:
Setpoint a
Drawto b
Drawto c

Case 3: One Contour
Through Parallel Edges



Expected Picture:
Setpoint a
Drawto b
Drawto c

Case 4: Contour Across
The Diagonal

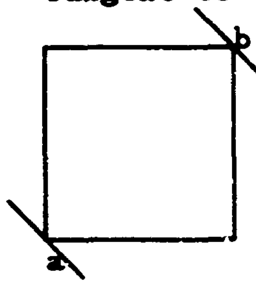


Expected Picture:
Setpoint a
Drawto b
Drawto c

Figure 1.4a
All Possible Contour Crossings of a 2 x 2 Subgrid

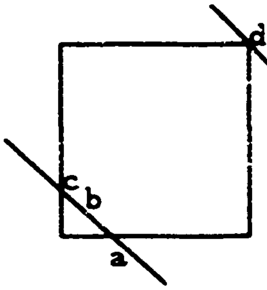
Double Contour Crossings

Case 5: Two Contours
Tangent to the 2 x 2



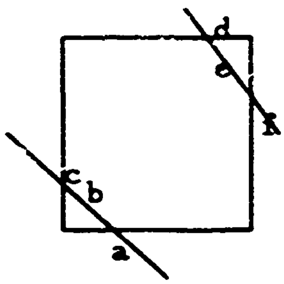
Expected Picture:
Drawpoint a
Drawpoint b

Case 6: One Contour Tangent,
One Contour through Adjacent Edges



Expected Picture:
Setpoint a
Drawto b
Drawto c
Drawpoint d

Case 7: Two Contours
Through Adjacent Edges

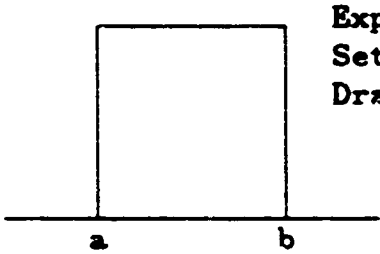


Expected Picture:
Setpoint a
Drawto b
Drawto c
Setpoint d
Drawto e
Drawto f

Figure 1.4b (continued)
All Possible Contour Crossings of a 2 x 2 Subgrid

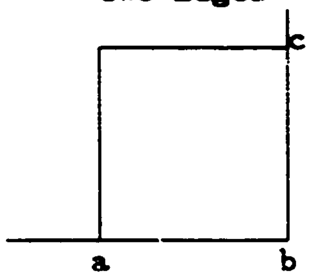
Equivalued Edges at the Contour Level

Case 8: Contour Along
One Edge



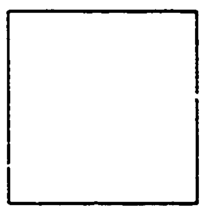
Expected Picture:
Setpoint a
Drawto b

Case 9: Contour Along
Two Edges



Expected Picture:
Setpoint a
Drawto b
Setpoint b
Drawto c

Case 10: Constant 2 x 2



Expected Picture:
None.

Figure 1.4c (continued)
All Possible Contour Crossings of a 2 x 2 Subgrid

the *Contour Tangent To The 2x2 case*. which is an even simpler case for connectivity generation.

It is not until we consider the two contours crossing the subgrid cases of *Figure* 1.4 that we realize the potential for problems with the above single contour crossing procedure. A procedure based only on connecting edge intersections cannot differentiate between cases such as the *Two Contours Tangent To The 2x2*, and the *Contour Across The Diagonal* cases. There are other similar connectivity generation problems evident for the two contours crossing cases. The *Two Contours Through Adjacent Edges* case has four edge intersections. For that case, information needs to be provided to the connectivity generation procedure that determines which of three possible intersection pairs should be connected.

Now that we have established a background for the connectivity problem for contour crossing of the 2x2 subgrid, we can detail the procedure used to solve that problem. Before we describe that algorithm, we first briefly review some of the problems cited in the literature for two-dimensional contouring.

C. TWO-DIMENSIONAL CONTOURING PROBLEMS

The literature on two-dimensional contouring, and the use of two-dimensional contouring for creating a contour surface display is extensive, encompassing a number of fields [Refs. 2-5], [Refs. 7-17]. A thorough review of the historical development of two-dimensional contouring algorithms and the properties of those algorithms is found in [Ref. 15]. Many of the contouring algorithms presented in that study are flawed either in that they generate an incorrect picture for some contour crossing cases, or in that they require special handling for "problem" 2x2 subgrids. Some of the typical algorithm problems detailed are identical to those described above, i.e. they concern *degenerate points*, where there are ambiguities as to which points to connect. In all of the algorithms reviewed, no attempt is made to fit the special cases inside of a general algorithmic framework. This is quite evident for the subgrid having a saddle point. That contour crossing case is handled by selecting the two lines "for which the direction changes the least" when compared against neighboring subgrids [Ref. 15]. Again, this requires special algorithmic resolution. None of the papers attempts to build a general framework useful for the generation of the coordinates and drawing instructions for any 2x2 subgrid. The following section describes both a data structure, the contouring tree, and an algorithm for using that data structure, that provide both a coherent framework for 2x2 subgrid contouring, and a comprehensive resolution to the 2x2 subgrid crossing problem.

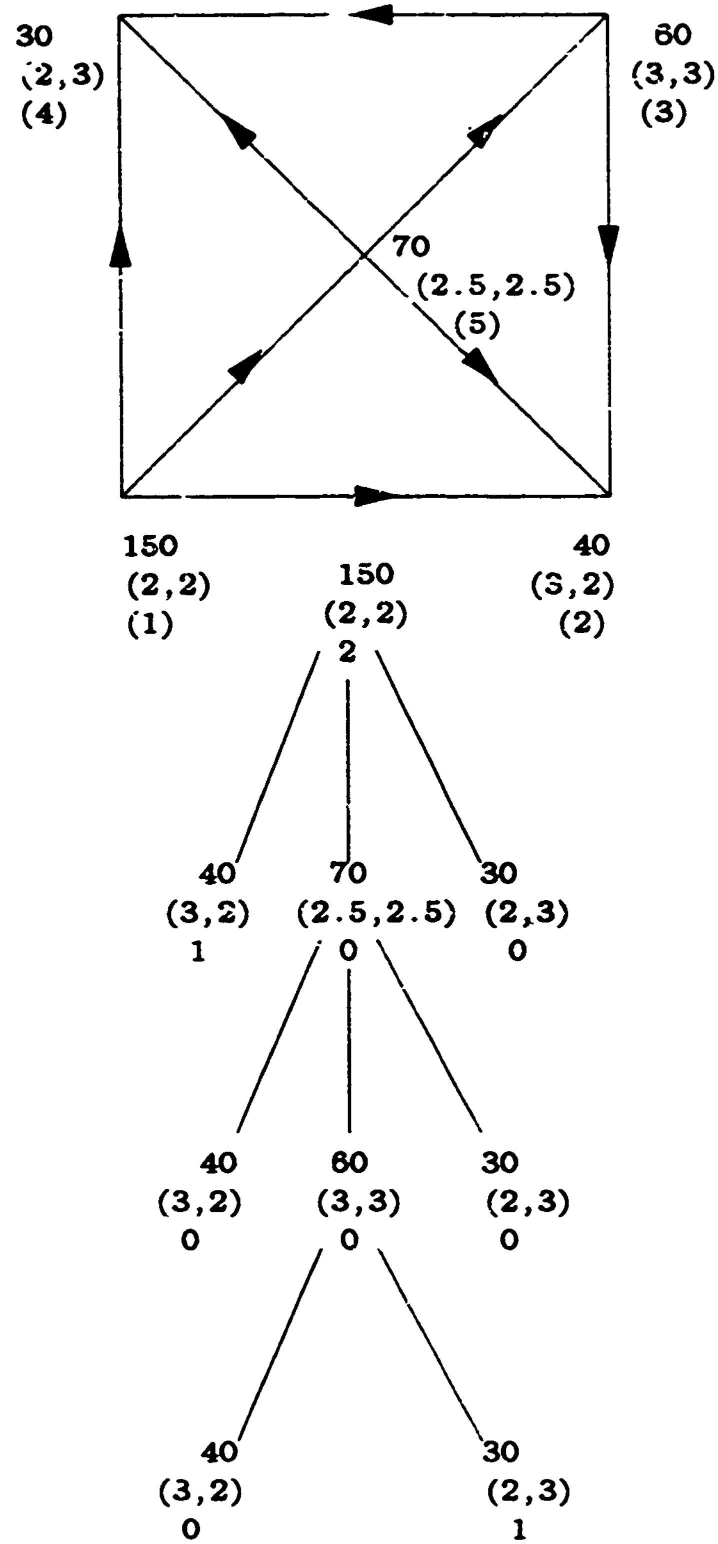


Figure 1.5a
Sample Contouring Tree for a 2 x 2 Subgrid

Tree Rooted At Value 150.00			
Level 50			
X	Y	Z	D
2.9091	2.0000	1.0000	1
2.8333	2.1667	1.0000	0
3.0000	2.5000	1.0000	0
2.6667	3.0000	1.0000	1
2.2500	2.7500	1.0000	0
2.0000	2.8333	1.0000	0

Tree Rooted At Value 150.00			
Level 100			
X	Y	Z	D
2.4545	2.0000	1.0000	1
2.3125	2.3125	1.0000	0
2.0000	2.4167	1.0000	0

Column D is the drawing command, ie. 1 = SETPOINT, 0 = DRAWTO.

Figure 1.5b

Coordinates Generated for Sample 2x2 Subgrid

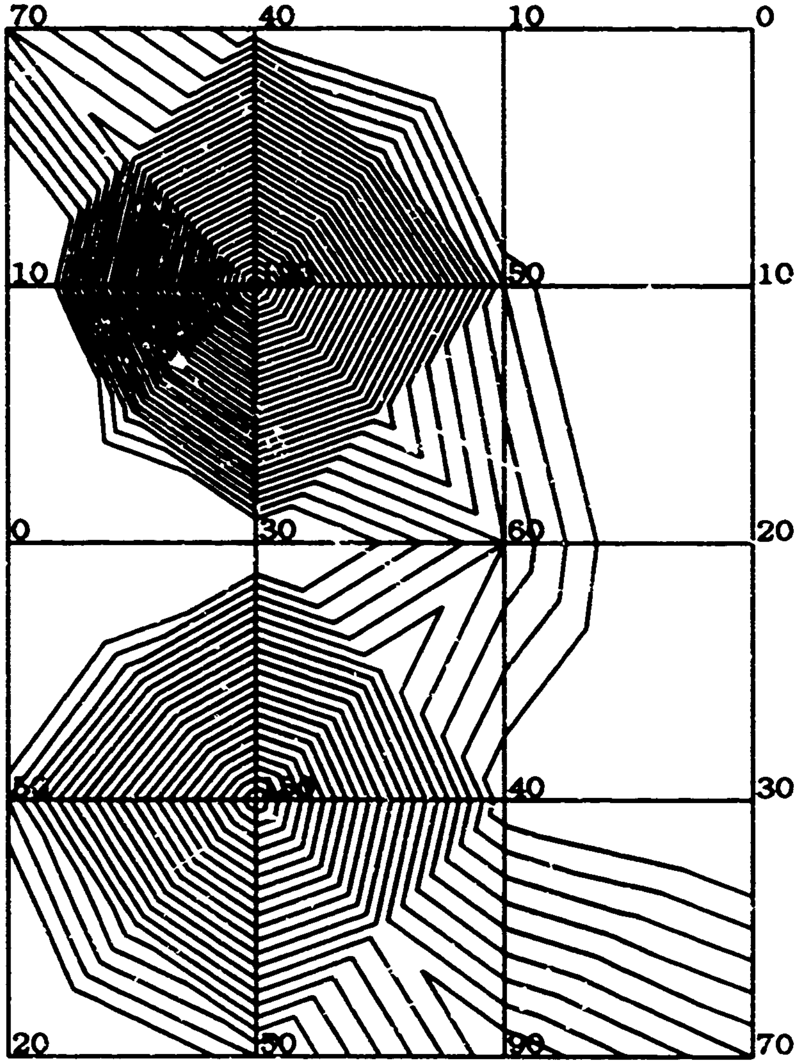
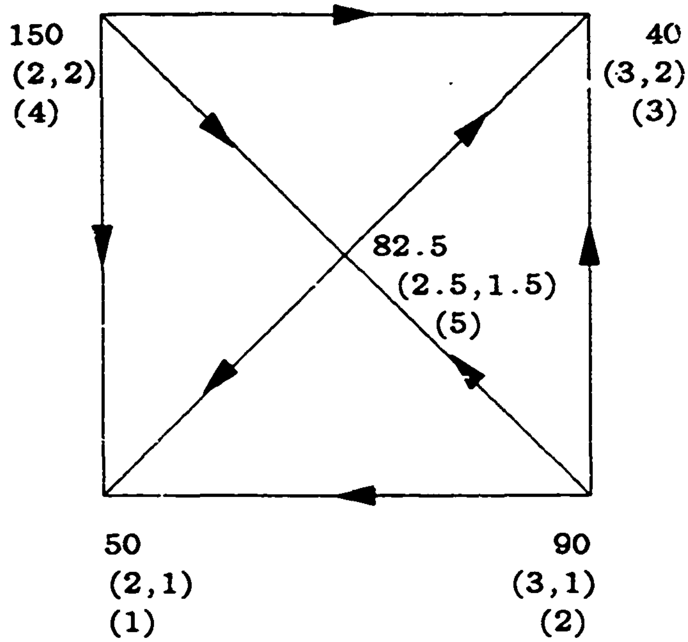


Figure 1.6
Example Contour Grid with Contours Drawn for Multiple Contour Levels



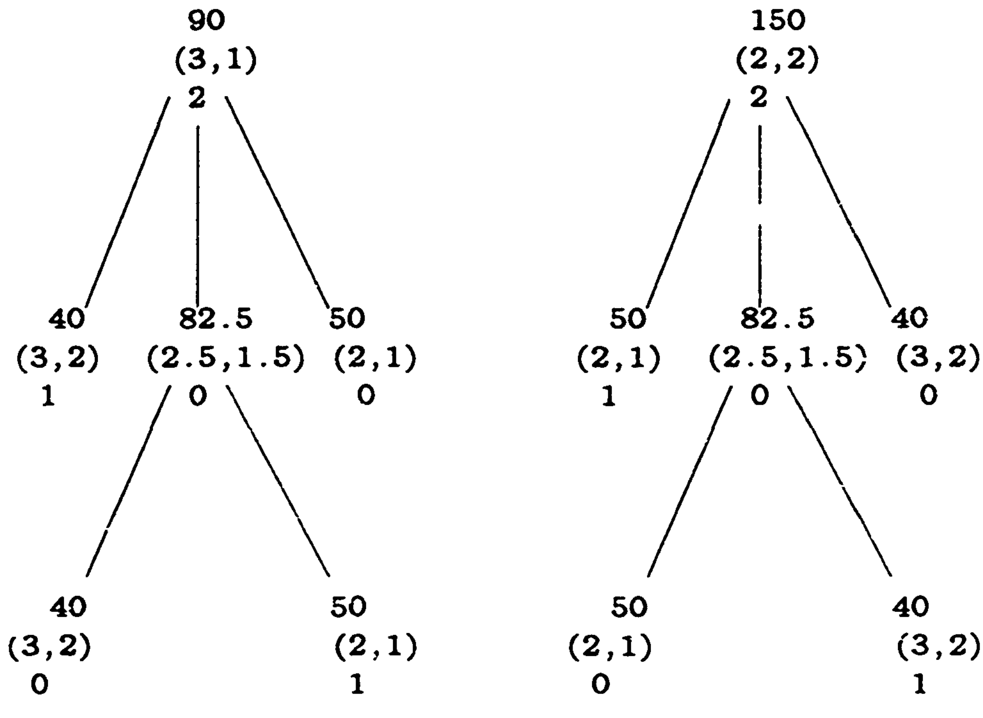


Figure 1.7a
Sample Contouring Tree for a 2 x 2 Subgrid with Saddle Point

First Tree Rooted At Value 90.00			
Level 50			
X	Y	Z	D
3.0000	1.8000	1.0000	1
2.8824	1.8824	1.0000	0
2.0000	1.0000	1.0000	1
2.0000	1.0000	1.0000	0

First Tree Rooted At Value 90.00			
Level 100			
X	Y	Z	D
no coordinates generated			

Second Tree Rooted At Value 150.00			
Level 50			
X	Y	Z	D
2.0000	1.0000	1.0000	1
2.0000	1.0000	1.0000	0
2.8824	1.8824	1.0000	1
2.9091	2.0000	1.0000	0

Second Tree Rooted At Value 150.00			
Level 100			
X	Y	Z	D
2.0000	1.5000	1.0000	1
2.3704	1.6296	1.0000	0
2.4545	2.0000	1.0000	0

Column D is the drawing command, ie. 1 = SETPOINT, 0 = DRAWTO.

Figure 1.7b

Coordinates Generated for Sample 2 x 2 Subgrid with Saddle Point

D. THE CONTOURING TREE

A contouring tree is a data structure that represents the edge value relationships of a 2x2 subgrid in a form that permits the rapid generation of the contour display for any contour level contained within the represented subgrid (see *Figure* 1.5), The formulation of the contouring tree is based upon the observation that for any two-dimensional grid a continuous series of contour displays can be created for contour levels in the range of the minimum and maximum grid values (see *Figure* 1.6, and [Refs. 1-5].

The use of the contouring tree is outlined best with an example of a small two-dimensional grid. *Figures* 1.2 and 1.3 depict the contours generated for contour level 50 and 100. The contours at level 100 are closed contours, forming simple, connected loops. The contours at level 50 are open contours. *Figures* 1.5 and 1.7 present the contouring trees created for two 2x2 subgrids of the 4x5 plane. The edges of the contouring trees correspond to the directed, downhill edges inscribed on the 2x2 subgrids of the figures. There are eight directed edges on each subgrid, four for the boundary edges and four for the edges to the subgrid's center point. The value used for the center point is the average of the four values comprising the corners of the 2x2 subgrid. (A reference as to the usefulness of the center point average value in generating smooth contours is found in [Ref. 15].) The edges of the contouring trees are ordered, maintaining the same counterclockwise ordering as in the original subgrids. A "1" under a node indicates that a **Setpoint** display command should be generated for any coordinate that is created along an edge that has that connectivity on its lower valued node. A "0" indicates a **Drawto** display command in a similar fashion and a "2" indicates a **Drawpoint**.

Display generation from a contouring tree is accomplished by performing a pre-order traversal of that contouring tree, producing a coordinate and drawing instruction whenever the desired contour level is found to be within the range of an edge of the contouring tree. A pre-order traversal visits the root, the left subtree, the middle subtree, and then the right subtree. An edge's range is defined to be the set of values between those associated with the nodes on either end of the edge. More precisely, we say a contour level is within an edge if the following condition holds:

lower_node's_value <= contour_level < higher_node's_value

For example, in *Figure* 1.5a at contour level 100, we issue coordinates and drawing instructions for the edges (2,2)-(3,2), (2,2)-(2.5,2.5), and (2,2)-(2,3). The drawing instruction issued for each of these edges is again the one associated with the lower valued node of the edge. The coordinate for each of these edges is generated by a linear interpolation of the edge's endpoint coordinates according to the decrease in contour level along the edge. The coordinates and drawing instructions generated for the contouring trees of *Figures* 1.5a and 1.7a are represented in *Figures* 1.5b and 1.7b.

There are some subtleties not evident from the above that are best detailed using a pseudocode description of the traversal algorithm. *Figure* 1.8 depicts the traversal procedure for the contouring tree assuming a particular data organization. The notation is quite standard. The pointers to the descendent nodes of NODE are LEFT(NODE), MIDDLE(NODE), and RIGHT(NODE). For each node of the contouring tree, there are three pieces of information: the value associated with the node, VALUE(NODE), the coordinate associated with the node, XYZ(NODE),and the connectivity associated with the node, CONN(NODE).

The generation of coordinates and drawing instructions from a contouring tree begins with routine CONTOUR_SUBGRID of *Figure* 1.8. That routine receives a pointer to the root node of the contouring tree. It then starts the traversal by calling routine VISIT with that root node. Routine VISIT checks to see if the edge defined by the passed in node and that node's ancestor, NODE and ANCESTOR, contains the contour level. If the edge does contain the contour level, the edge intersection coordinate is computed using linear interpolation and issued to the display along with the connectivity associated with that node, CONN(NODE). If we issue a coordinate and connectivity for a node, we need to check the subtree under that node for equivalued edges. If an equivalued edge at the contour level is found, a coordinate and drawing instruction pair are issued for that equivalued edge (routine VISIT_SUBTREE). Once a coordinate and

Contouring Tree Description

Pointers to descendent nodes:

```
LEFT(NODE)
MIDDLE(NODE)
RIGHT(NODE)
```

Values associated with each node:

```
VALUE(NODE): grid value
XYZ(NODE): coordinate of that grid value.
CONN(NODE): drawing instruction.
```

```
procedure CONTOUR_SUBGRID(ROOT)

      VISIT(ROOT.ROOT) #begin the traversal of the pointed at
                       # contouring tree.

end.


Procedure VISIT(NODE , ANCESTOR)

      if (NODE == NULL)
      {
            return
      }

      if((VALUE(NODE) <= CONTOUR_LEVEL < VALUE(ANCESTOR))
            OR
        (VALUE(NODE)==CONTOUR_LEVEL AND NODE==ANCESTOR))
      {

            #Edge contains the contour level.

            Issue a coordinate computed via linear interpolation
            along the edge.

            Issue CONN(NODE) as the drawing instruction.
```

Figure 1.8

Pseudocode of the Traversal Algorithm for the Contouring Tree

```
            # Check subtrees of this node for equivalued edges.
            VISIT_SUBTREE(LEFT(NODE),NODE)
            VISIT_SUBTREE(MIDDLE(NODE),NODE)
            VISIT_SUBTREE(RIGHT(NODE),NODE)

            return # no need to examine the subtree further.

        } # endif coordinates were generated for an edge.

        VISIT(LEFT(NODE),NODE) # visit left subtree.
        VISIT(MIDDLE(NODE),NODE) # visit middle subtree.
        VISIT(RIGHT(NODE),NODE) # visit right subtree.

        return

end


Procedure VISIT_SUBTREE(SUBNODE,SUBANCESTOR)
    if(SUBNODE == NULL)
    {
        return
    }

    if(VALUE(SUBNODE) == CONTOUR_LEVEL)
    {

        Issue coordinates for the equivalued edge.
        Setpoint on XYZ(SUBANCESTOR).
        Drawto XYZ(SUBNODE).

    }

    VISIT_SUBTREE(LEFT(SUBNODE),SUBNODE)
    VISIT_SUBTREE(MIDDLE(SUBNODE),SUBNODE)
    VISIT_SUBTREE(RIGHT(SUBNODE),SUBNODE)

    return
end
```

Figure 1.8 (continued)

Pseudocode of the Traversal Algorithm for the Contouring Tree

drawing instruction pair have been issued for an edge, and once the subtree beneath that edge has been investigated for equivalued edges, further traversal of that subtree is terminated. If an edge is found not to contain the contour level, the traversal continues as depicted at the bottom of routine VISIT.

The pre-order traversal procedure described above generates the coordinates and drawing instructions for the part of the 2x2 subgrid the contouring tree represents. To generate the coordinates for a larger two-dimensional grid, we generate the contouring trees for each 2x2 subgrid of that grid and then apply the traversal procedure to those trees. We note here that no ordering is required in the generation of coordinates for the 2x2 subgrids.

E. PROBLEMS WITH THE 2X2 SUBGRID ALGORITHM

Having presented the use of the contouring tree, we must look back and discuss its capabilities and limitations. The initial impression is that the contouring tree provides a nice, uniform framework for generating the coordinates and drawing instructions appropriate to the 2x2 subgrid. The algorithm also takes care of the difficult two contours crossing case for the 2x2 subgrid. The algorithm correctly handles subgrids containing equivalued lines at the contour level. The algorithm also handles subgrids containing a single grid point at the contour level.

The core problems with this algorithm all concern issues of picture efficiency. Since the display generated for each 2x2 subgrid is generated independently of any neighboring 2x2 subgrids, equivalued lines at the contour level on the border of a subgrid will be duplicated. A similar problem occurs for subgrid corner values that equal the contour level. If we display either of the above cases on a calligraphic display device, we will see a bright line for the equivalued edge, and a bright point for the grid value equal to the contour level. Another problem, also due to the independent computation of each 2x2 subgrid, is that no ordering is provided for coordinates that come out of this algorithm. For calligraphic displays, this is a problem because for such devices electron beam movement is expensive. A contour display that causes the maximum movement of the electron beam every other subgrid greatly decreases the vector capability of the calligraphic display device.

II. THE NEW LARGE CONTOURING TREE ALGORITHM

The 2x2 Subgrid Algorithm builds a general framework useful for the generation of the coordinates and drawing instructions for any 2x2 subgrid. But as described above, there is a picture efficiency problem with this algorithm, i.e. edge duplication and vector ordering problems. We have developed a new algorithm, called the **Large Contouring Tree Algorithm**, that solves these problems.

A. OVERVIEW OF THE NEW ALGORITHM

The new algorithm generates contours for two-dimensional, rectangular grids, i.e. grids composed of multiple 2x2 subgrids. In this chapter, we use a 3x3 grid for our examples of the component parts of the algorithm (see *Figure* 2.1). This grid is a portion of the grid shown in *Figure* 1.2.

The input data to the Large Contouring Tree Algorithm is the size of the grid, the density values in the grid, and the contour level. The output of the algorithm is the set of the coordinates and drawing instructions representing the chosen contour level. *Figure* 2.2 shows the contours generated for the sample subgrid of *Figure* 2.1 for contour levels 50 and 100.

To generate contours, the Large Contouring Tree Algorithm goes through the following steps. The first step of the algorithm is to calculate the density values of the center points of each 2x2 subgrid of the larger grid. (A reference as to the usefulness of the center point of average value in generating smooth contours is found in [Ref. 15].) *Figure* 2.3 shows the sample grid with the calculated average density values for the center points on the grid.

The second step is the creation of the directed graph from the grid using the density values. We create the directed graph by assigning a direction to each edge of the grid based upon the values assigned to each node of the grid. Equivalued edges are assigned an arbitrary direction.

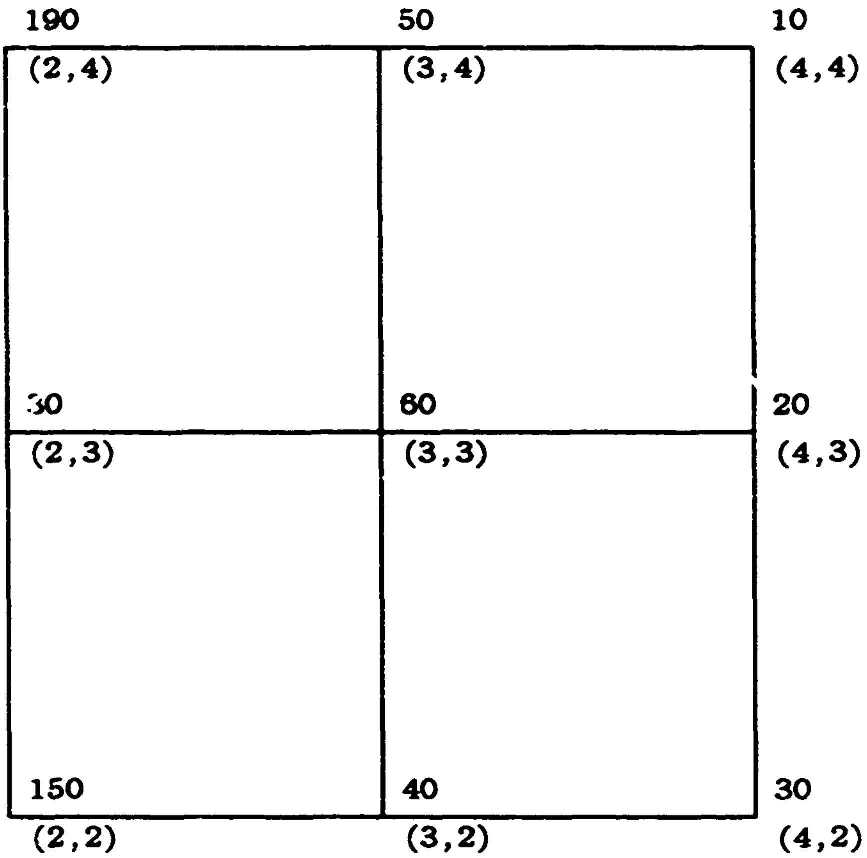


Figure 2.1
The Sample Grid Taken From The Grid Of Figure 1.2

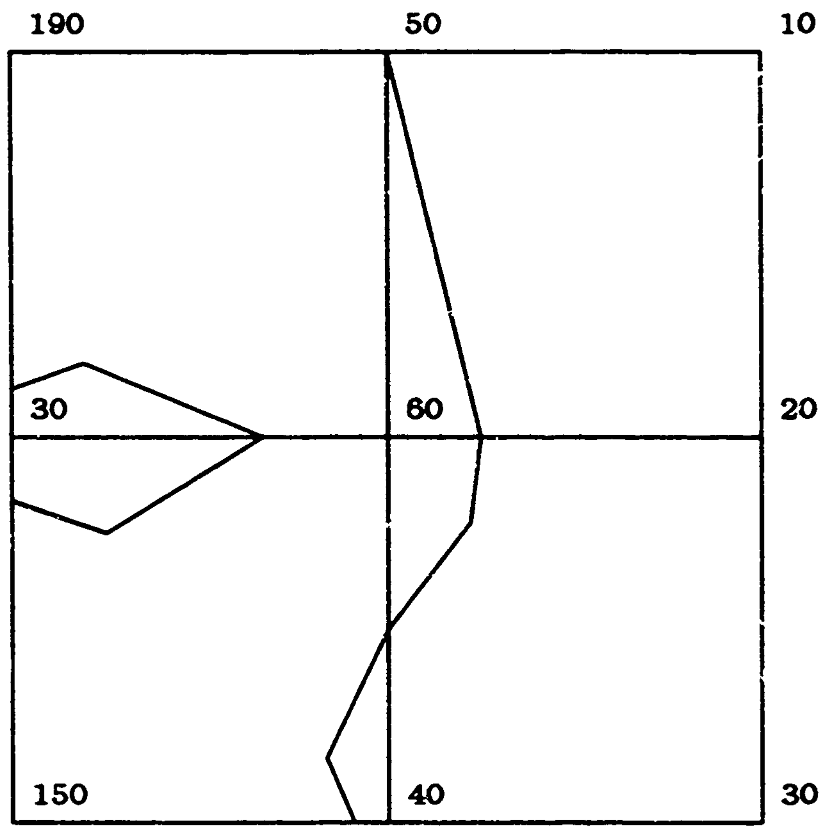


Figure 2.2a
The Contours Generated For The Sample Grid At Contour Level 50

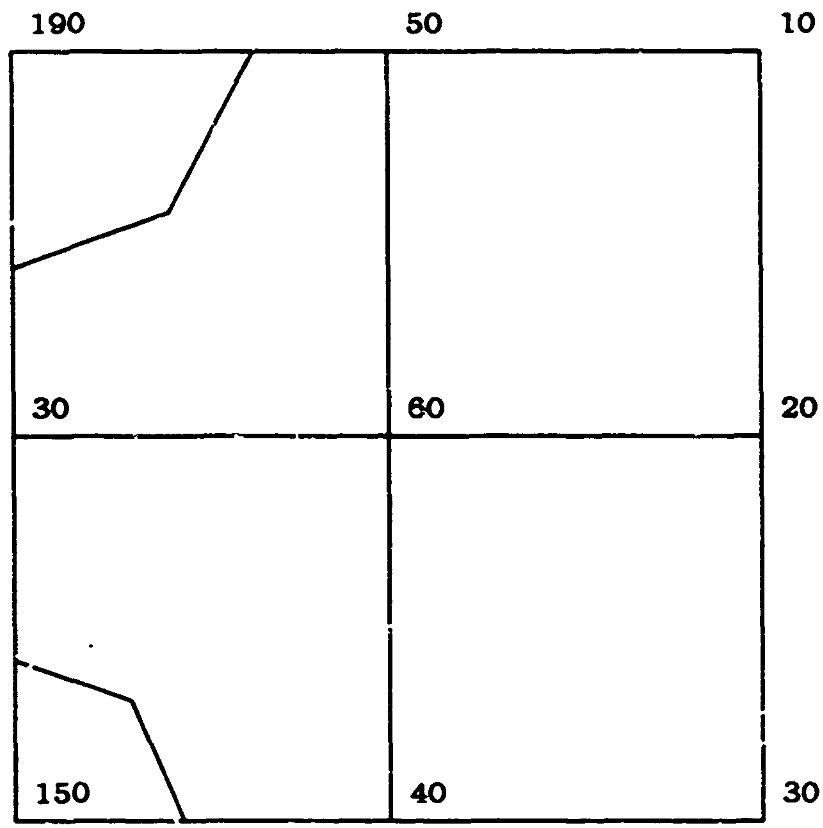


Figure 2.2b
The Contours Generated For The Sample Grid At Contour Level 100

The next step is the creation of the in-degree matrix of the directed graph. The in-degree matrix of a directed graph is defined in [Ref. 18]. We provide more detail about in-degree matrix creation in the following section.

We create a contouring tree for each node of the in-degree matrix that has the property in-degree(i,i)=0, i.e. for each node lacking incoming edges. Contouring tree creation is accomplished in a manner similar to that used for the 2x2 subgrid, i.e. we construct directed trees from the information contained in the in-degree matrix, making sure that the order of edge attachment in the tree corresponds to the order in the directed graph. The difference between the 2x2 subgrid algorithm and the large contouring tree algorithm is that the tree construction process of the large contouring tree algorithm is not limited to a single 2x2 subgrid but rather is allowed to extend to multiple subgrids.

After contouring tree creation for the two-dimensional grid, the next step in the tree construction process is to associate drawing commands with each of the trees' nodes. Drawing commands are placed in the contouring tree to indicate when a line enters the region represented by the contouring tree either from a neighboring subgrid, or from a location off of the grid. We insert the drawing commands by way of a pre-order traversal of the contouring tree, placing a setpoint command on each node that is a new lowest value for the tree. There are other more detailed considerations with respect to drawing command placement. These are discussed below.

Once the contouring tree for a two-dimensional grid has been constructed, and once the drawing commands have been placed into that structure, the contouring tree is ready for use in generating a contour display. Display generation is accomplished by performing a pre-order traversal of the contouring tree, producing a coordinate and drawing instruction whenever the desired contour level is found to be within the range of on edge of the contouring tree. If an equivalued edge at the contour level is found, a coordinate and drawing instruction pair are issued for that equivalued edge.

B. CONTOURING TREE CREATION

Contouring tree construction is best understood if we describe that procedure in graph theoretic terms. The first step in that procedure is to create all of the nodes that take part in the directed graph. This set of nodes consists of all grid crossing points and the set of nodes corresponding to the center points of average value for each individual 2x2 subgrid of the larger two-dimensional grid. *Figure* 2.3 shows the sample 3x3 grid with the center points of average value labeled.

Once we have the collection of nodes for the directed graph, we then need to assign directed edges between those nodes. Edges are directed from nodes of high value towards nodes of low value. The edge connections between each node correspond to their connection in the original two-dimensional grid. Equivalued edges are assigned directions arbitrarily. For example, the grid in *Figure* 2.3 has thirteen nodes in its directed graph, counting the grid crossing nodes and the center points of average value. *Figure* 2.4 shows this same grid with arrowheads indicating edge directions.

Once we have the directed graph corresponding to the two-dimensional grid, the question then becomes, how do we obtain the contouring tree, or trees from the directed graph? We can put this question in terms of graph theory if we notice that a contouring tree is a directed tree. The problem then becomes one of obtaining the directed tree, or trees, from the directed graph such that the order of edge attachment in the tree corresponds to the order in the directed graph. From graph theory, we have the requirement that a directed tree has the in-degree of its root node equal to zero, and the in-degree of every other node equal to one [Ref. 18]. To examine the in-degree of each node of the directed graph, we must construct the in-degree matrix D for that graph. The in-degree matrix D of a directed graph G is defined in [Ref. 18] as :

D(i,j) = in-degree(i), if i=j
D(i,j) = -k, if i is not equal to j,
where k is the number of
edges in G from i to j
(i.e., -1 for all our graphs).

Figure 2.6 shows the in-degree matrix for the directed graph of *Figure* 2.4. The node numbering scheme of *Figure* 2.5 is used for the in-degree matrix of *Figure* 2.6. From *Figure* 2.6, we note that the roots of the contouring trees are

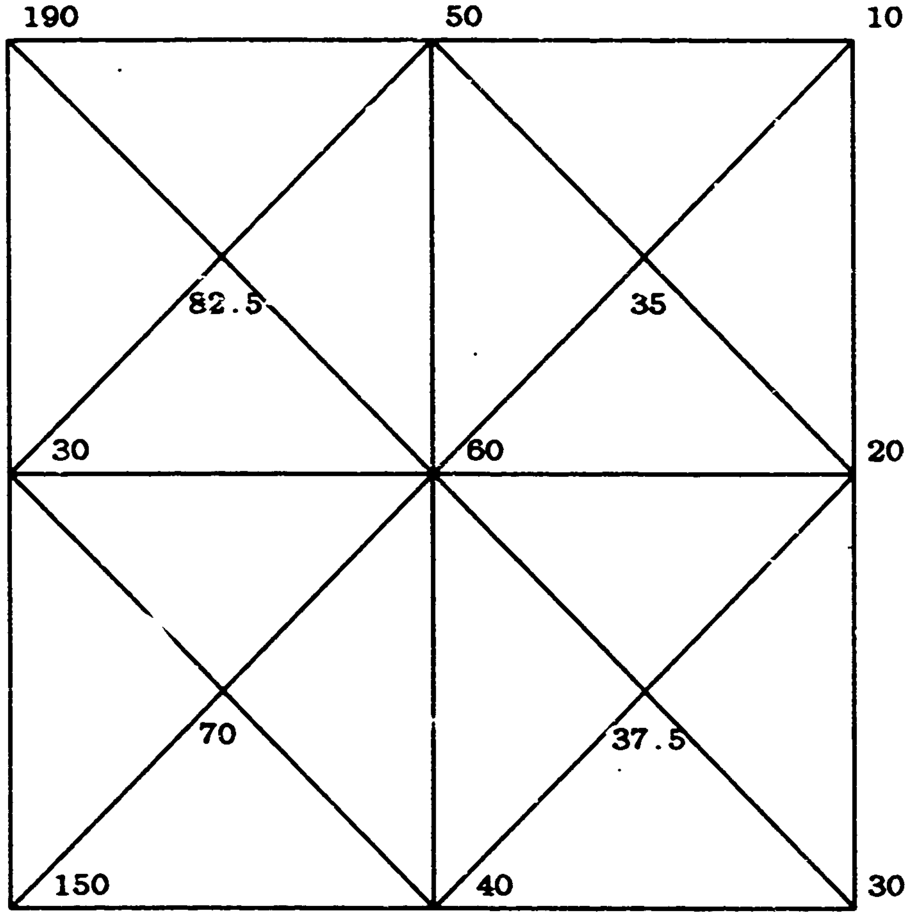


Figure 2.3
The Sample Grid With The Calculated Average
Density Values For Center Points

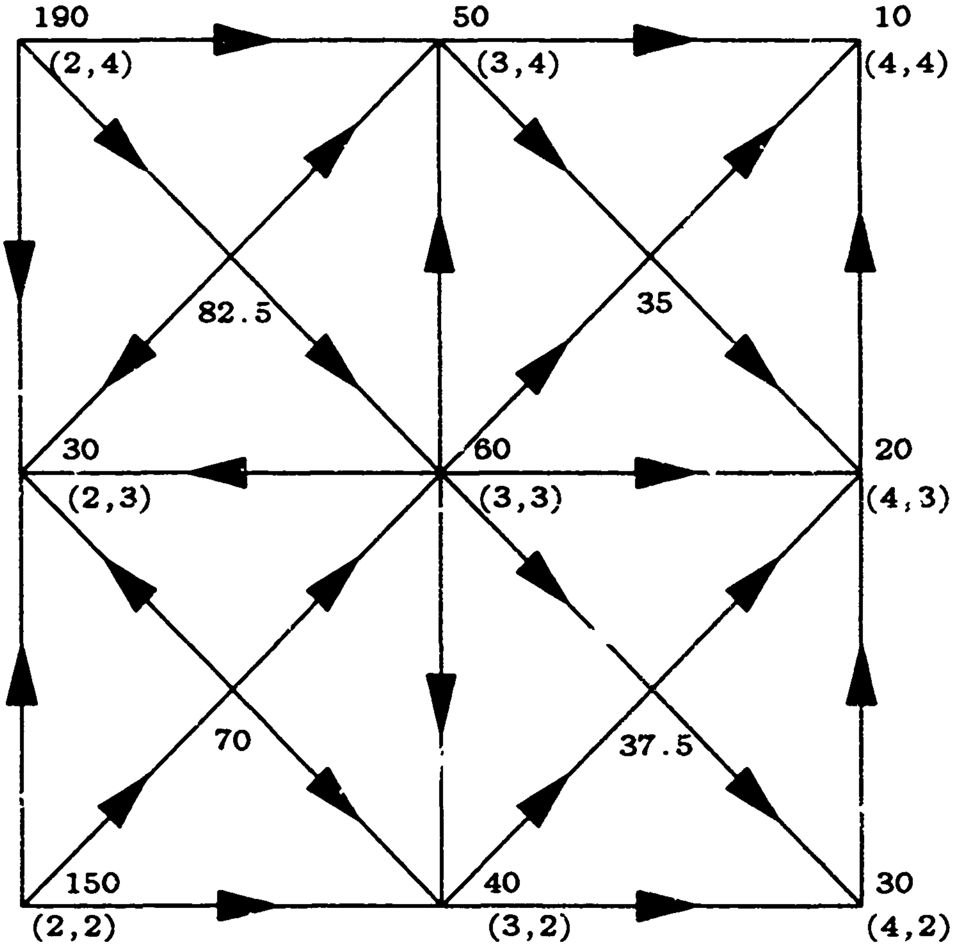


Figure 2.4
The Directed Graph Created From The Sample Grid

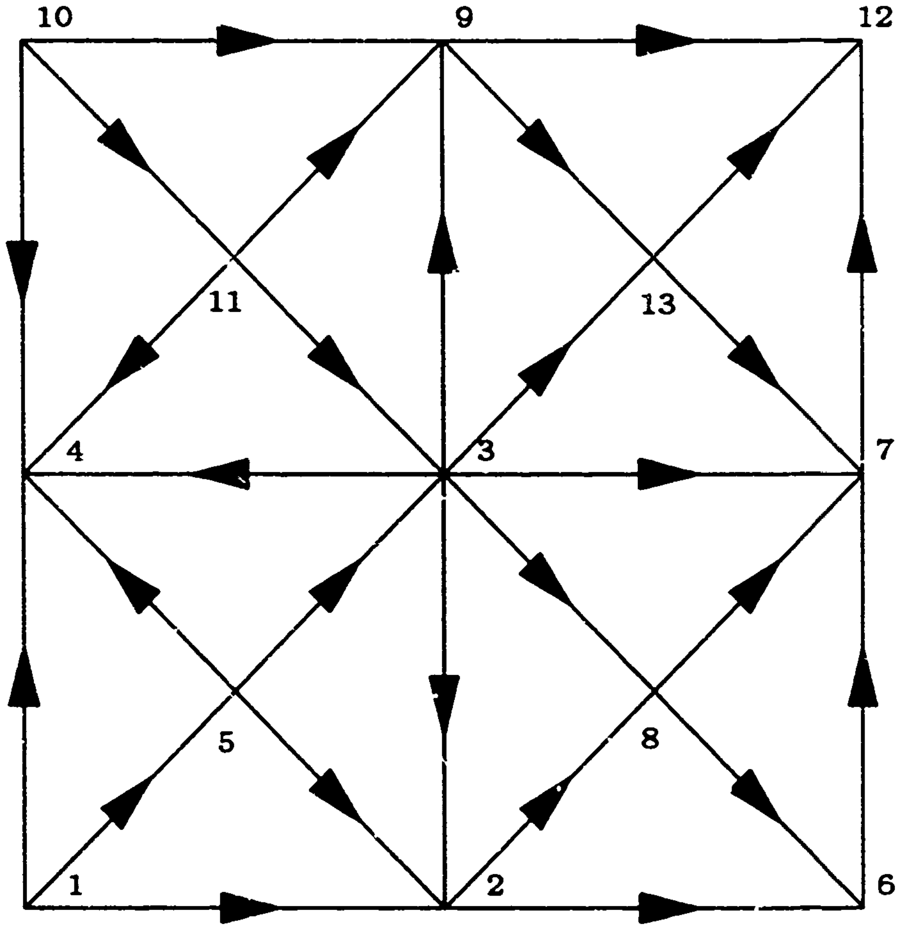


Figure 2.5
The Numbers On The Directed Graph Represent The Node Number Associated With The Node In The In-Degree Matrix

recognizable from the in-degree matrix as D(i,i)=0. This matches the first part of the directed tree requirement. For multiple roots (in-degree(v)=0 for more than one node), we create as many trees as the number of roots in the in-degree matrix. For each diagonal entry D(i,i)=n, where n > 1, we create n-1 duplicates of that node, for a total of n, taking care to copy the appropriate values, coordinates, etc. We then reassign the original edges that went to the single node, such that each edge receives its own copy of the duplicated node.

1. In-Degree Matrix Creation

The first problem we encounter once we have a two-dimensional, regular grid, and its center nodes of average value, is how to create the in-degree matrix. We solve that problem by enumerating all possible configurations of edges that could form such a grid. We call each of these configurations, or cases, situations. For example, Situation 4 is the name of the case for the node in the center of a 2x2 subgrid. We locate and classify all of the nodes and edges of the original grid into one of ten possible situations. We have a number/name and characteristic assigned to each situation. Some situations are comprised of only one node in the grid, and some include multiple nodes. Each node in the grid belongs to one of these situations. For example, the node in the upper righthand corner of *Figure* 2.7 is a Situation 10 grouping, i.e. it is comprised of the upper righthand node of the two-dimensional, grid. The complete set of situations are defined as follows.

Situation 1 is the name of the case for the node in the lower lefthand corner of the grid. Situation 3 is the name of the case for the node in the lower righthand corner of the grid. Situation 8 is the name of the case for the node in the upper lefthand corner, and Situation 10 is the name of the case for the node in the upper righthand corner. Situation 4 is the name of the case for the node in the center of a 2x2 subgrid. *Figure* 2.7 shows a 2x2 subgrid which is a combination of situations 1,3,4,8, and 10.

All nodes on the perimeter grid line with the lowest valued Y coordinate, except for the first and last nodes, are named Situation 2 nodes. All nodes on the perimeter grid line with the highest valued Y coordinate, except for the first and last nodes, are named Situation 9 nodes. All nodes on the perimeter grid line with the lowest valued X coordinate, except for the first and last nodes, are named

D(i,j)	1	2	3	4	5	6	7	8	9	10	11	12	13
1	0	-1	0	-1	-1	0	0	0	0	0	0	0	0
2	0	3	0	0	0	-1	0	-1	0	0	0	0	0
3	0	-1	2	-1	0	0	-1	-1	-1	0	0	0	-1
4	0	0	0	5	0	0	0	0	0	0	0	0	0
5	0	-1	-1	-1	1	0	0	0	0	0	0	0	0
6	0	0	0	0	0	2	-1	0	0	0	0	0	0
7	0	0	0	0	0	0	4	0	0	0	0	-1	0
8	0	0	0	0	0	-1	-1	2	0	0	0	0	0
9	0	0	0	0	0	0	0	0	3	0	0	-1	-1
10	0	0	0	-1	0	0	0	0	-1	0	-1	0	0
11	0	0	-1	-1	0	0	0	0	-1	0	1	0	0
12	0	0	0	0	0	0	0	0	0	0	0	3	0
13	0	0	0	0	0	0	-1	0	0	0	0	-1	2

Figure 2.6

Sample In-Degree Matrix For The Directed Graph Of Figure 2.3

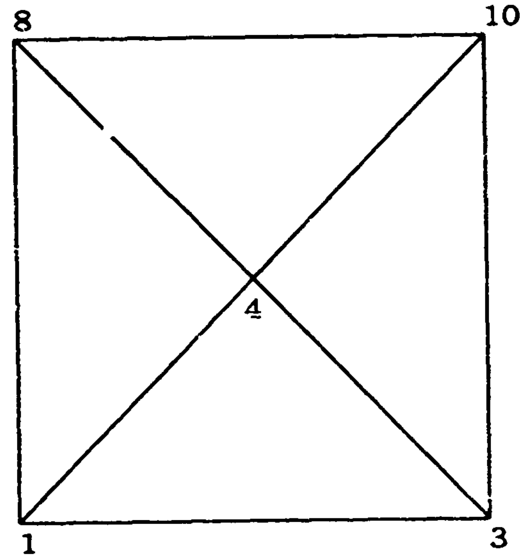


Figure 2.7
Grid Nodes Are Named By Their Situation Number

Situation 5 nodes. All nodes on the perimeter grid line with the highest valued X coordinate, except for the first and last nodes, are named Situation 7 nodes. Non-perimeter nodes that occur at crossing points of the two-dimensional grid, i.e. non-center point of average value nodes, are named Situation 6 nodes. *Figure* 2.8 shows all ten possible grid node name situations. *Figure* 2.9 is a summary of those situations.

Once we have a mechanism for naming each of the possible node configurations for a two-dimensional grid, we then need to provide a node visitation naming scheme that allows the association of a node in the two-dimensional grid with its record in the in-degree matrix. We call this naming scheme the **Visiting Node Order.** The numbers on *Figure* 2.10 show this visiting node order for the example grid. We begin by visiting the first node (1.1), then the second node (2.1), and then the other nodes respectively (2.2), (1.2),

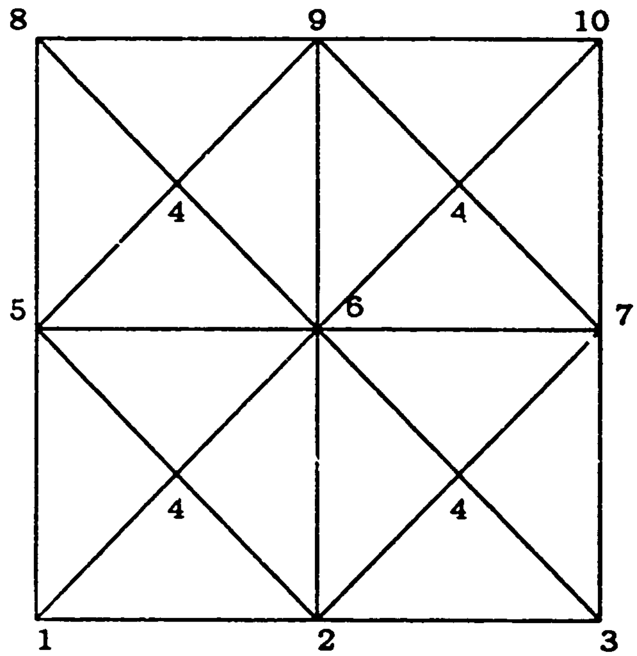


Figure 2.8
All Ten Possible Grid Node Name Situations

(1.5,1.5), etc. What we mean here is that the first node (1,1) is associated with the node number 1 in the in-degree matrix. The second node (2,1) in the grid is associated with the node number 2 in the in-degree matrix. The third node (2,2) is associated with the node number 3 in the in-degree matrix, and so on. According to our rule, we visit every node in the 2x2 subgrid before going onto the next 2x2 subgrid. The same rule is then applied to the next 2x2 subgrid. Already visited nodes in the 2x2 subgrid are skipped. The rest of them are visited in counterclockwise order. If we exhaust all 2x2 subgrids in the X direction, then the next 2x2 subgrid to be visited is on top of the 2x2 subgrid which we visited first. Let's say that we are on the node number 8 of *Figure* 2.10. The next 2x2 subgrid to be visited is bounded by nodes (1,2), (2,2), (2,3), and (1,3). According to our rule, the first node we should visit is node (1,2), then node (2,2). But these nodes have been already visited. Given the rule above, we skip these nodes and

The Definition Of Situations

SITUATION 1 is the name of the case for the node in the lower lefthand corner of the grid.

SITUATION 2 is the name of the case for all nodes on the perimeter grid line with the lowest valued Y coordinate, except for the first and last nodes.

SITUATION 3 is the name of the case for the node in the lower righthand corner of the grid.

SITUATION 4 is the name of the case for the node in the center of a 2x2 subgrid.

SITUATION 5 is the name of the case for all nodes on the perimeter grid line with the lowest valued X coordinate. except for the first and last nodes.

SITUATION 6 is the name of the case for non-perimeter nodes that occur at crossing points of the two-dimensional grid.

SITUATION 7 is the name of the case for all nodes on the perimeter grid line with the highest valued X coordinate. except for the first and last nodes.

SITUATION 8 is the name of the case for the node in the upper lefthand corner of the grid.

SITUATION 9 is the name of the case for all nodes on the perimeter grid line with the highest valued Y coordinate. except for the first and last nodes.

SITUATION 10 is the name of the case for the node in the upper lefthand corner of the grid.

Figure 2.9

Summary of the Ten Possible Grid Node Name Situations

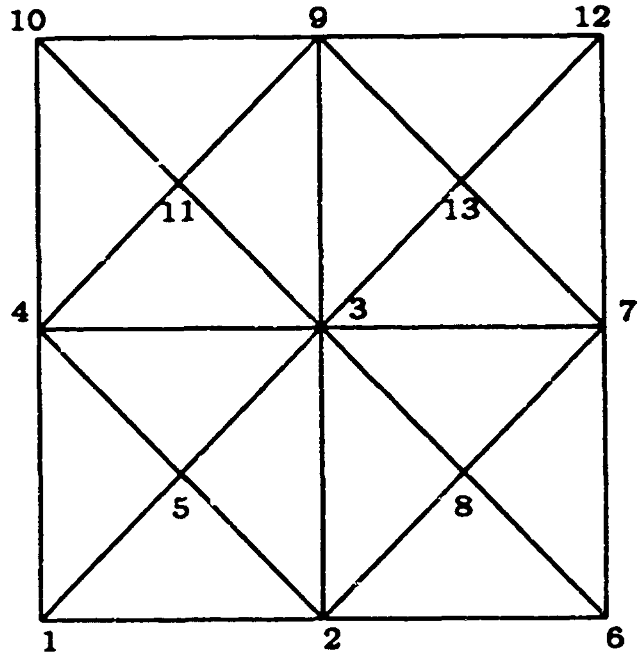


Figure 2.10
The Visiting Node Order Of The Sample Grid

then visit nodes (2.3). (1.3). and (1.5.2.5) respectively. The procedure is to first exhaust all 2x2 subgrids in the X direction. then go to the 2x2 subgrid (if there is one) on top of the first visited 2x2 subgrid. and then apply the same rule on the following 2x2 subgrids in the X direction. This process is repeated until there are no more 2x2 subgrids to be visited in the grid.

Visiting node order is the order in which the nodes of the two-dimensional grid and center points of average value are visited during in-degree matrix creation. The in-degree matrix procedure works in the following fashion as it steps through each node. First. the situation name for the node is determined.

The second step is to call a procedure that examines the edges attached to the node. The procedure called is selected on the basis of the situation name for the node. *Figure* 2.11 shows the pseudocode of the in-degree matrix creation procedure. *Figure* 2.6 holds the in-degree matrix for the directed graph of *Figure* 2.4.

2. Contouring Tree Construction

In the first chapter, we described the contouring algorithm for the 2x2 subgrid. The difference between the 2x2 subgrid algorithm and the large contouring tree algorithm is that the tree construction process of the large contouring tree algorithm is not limited to a single 2x2 subgrid but rather is allowed to extend to multiple subgrids. In this chapter, we show how to create the large contouring trees from the in-degree matrix. This creation operation is a "tree growth" process. We build each contouring tree by adding edges successively, starting at the root, and then recursively to each descendent node.

Above, we created the directed graph from the given grid and then built the in-degree matrix from this directed graph. The in-degree matrix and the directed graph provide the information necessary to create the contouring tree. The in-degree matrix reflects the direction of the edges in the directed graph. We use this feature of the in-degree matrix during the tree growth process. Before we describe the tree growth process, it is necessary to provide some background on how edges are related to the nodes of the grid.

As mentioned above, each node in the grid can be characterized as belonging to one of ten situations. The number of edges in each of the ten situations is constant. For example, situation 1 has three edges, 2 has five edges, etc. This constancy allows the creation of an edge list for each situation. This, in turn, allows the assignment of a number-name to each edge. The numbers assigned each edge represents the order used for edge addition in the tree growth process. This edge ordering is maintained in the contouring tree. Given this background, we then examin in detail contouring tree creation from the in-degree matrix.

The algorithm of *Figure* 2.13 outlines the general procedure for contouring tree construction. The first part of this procedure is a loop that starts

NOTE: The range of **MidNum** is the set of visiting node order numbers on the node in the center of each 2x2 subgrid. The range of **Xnum** is the set of visiting node order numbers for the rest of the nodes in the grid.

```
Procedure CREATE_IN_DEGREE_MATRIX (XMAX,YMAX)
    # XMAX and YMAX are the maximum values of X and Y coordinate

    MidNum <-- 5
    XNum <-- 1

    For X=1 to XMAX
    {
        For Y=1 to YMAX
        {
            If (X=1) AND (Y=1)
                EVALUATE_SITUATION_1 (X,Y,MidNum,XNum)

            Else If (X < XMAX) AND (Y = 1)
                EVALUATE_SITUATION_2 (X,Y,MidNum,XNum)

            Else If (X = XMAX) AND (Y = 1)
                EVALUATE_SITUATION_3 (X,Y,MidNum,XNum)

            Else If ( X > XMAX )
                EVALUATE_SITUATION_4 (X,Y,MidNum,XNum)

            Else If (X = 1) AND NOT (Y = YMAX)
                EVALUATE_SITUATION_5 (X,Y,MidNum,XNum)

            Else If NOT ((X = XMAX) OR (Y = YMAX))
                EVALUATE_SITUATION_6 (X,Y,MidNum,XNum)

            Else If (X = XMAX) AND NOT (Y = YMAX)
                EVALUATE_SITUATION_7 (X,Y,MidNum,XNum)

            Else If (X = 1) AND (Y = YMAX)
                EVALUATE_SITUATION_8 (X,Y,MidNum,XNum)

            Else If (X = XMAX) AND (Y = YMAX)
                EVALUATE_SITUATION_9 (X,Y,MidNum,XNum)

            Else EVALUATE_SITUATION_10 (X,Y,MidNum,XNum)

        } # Endfor
    } # Endfor

    - Find the diagonal values by counting the minus one values of the
    column in the in-degree matrix.

End    # procedure
```

Figure 2.11

Pseudocode Of Creation Of The In_Degree Matrix

```
Procedure EVALUATE_SITUATION_1 (X,Y.MidNum.XNum)
    # Check all the edges connected to the node
    # Check edge number 1
    If (NodeDnsty(X.Y) => NodeDnsty(X+1,Y))
        In-Degree(X.X+1) <-- -1
    else In-Degree(X+1.X) <-- -1
    # Check edge number 2
    If (NodeDnsty(X.Y) => NodeDnsty(XMAX+1,Y))
        In-Degree(X,MidNum) <-- -1
    else in-Degree(MidNum,X) <-- -1
    # Check edge number 3
    If (NodeDnsty(X,Y) => NodeDnsty(X,Y+1))
        In-Degree(X,MidNum-1) <-- -1
    else In-Degree(MidNum-1,X) <-- -1

    # Increments of MidNum and XNum

    XNum <-- XNum + 1
    If ( XMAX <> 2 )
        MidNum <-- MidNum + 3
end # Procedure

Procedure EVALUATE_SITUATION_<Number> (X.Y.MidNum.XNum)

     - Check all the edges connected to the node in situation<number>.
     - Put the results into the In_Degree Matrix. using "MidNum" and "XNum"
     - Issue the next value of "MidNum"
     - Issue the next value of "XNum"

End    # Procedure
```

Figure 2.11 (continued)

Pseudocode Of Creation Of The In_Degree Matrix

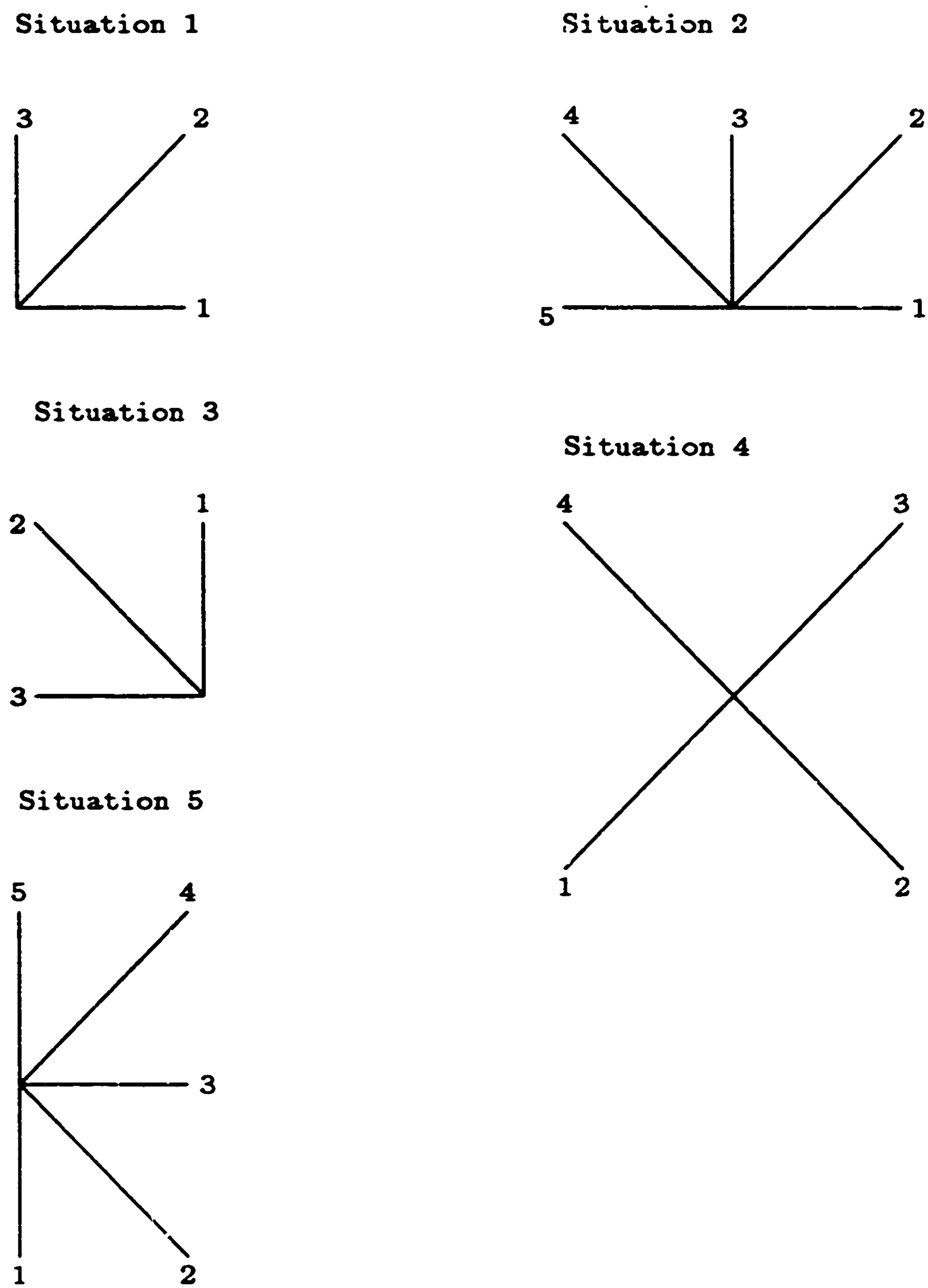


Figure 2.12
Counterclockwise Edge Order For Each Situation

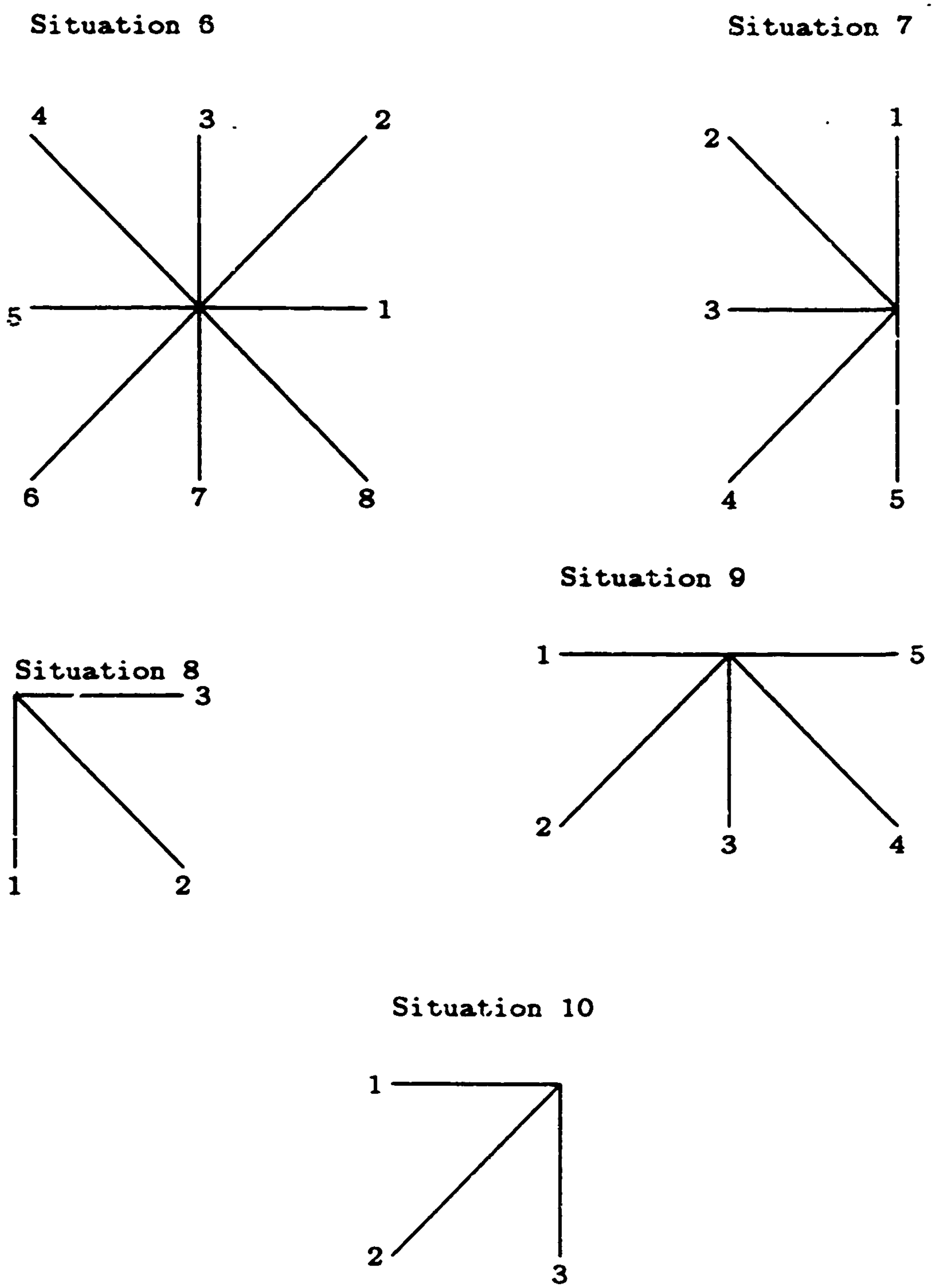


Figure 2.12 (Continued)
Counterclockwise Edge Order For Each Situation

```
Procedure CONSTRUCT_LARGE_CONTOURING_TREE ( MAX_SIZE )

  # MAX_SIZE is the maximum node number of the in-degree matrix

    For i = 1 to MAX_SIZE
    {

        If In_Degree(i,i) = 0    # zero value on diagonal means the root node
        {

            Growth_Node <-- Node(i)

            Repeat

            {
                    - Find which situation the growth node belongs to.
                    - Put the edges on the edge list corresponding to the
                    growth node into clockwise order (see Figure 2.12).
                    - Push the new ordered edge list onto the stack.

                Repeat

                    {

                        - Pop one edge from the top of the stack.

                    }

                Until ( Edge direction is outward from the growth node )
                                        OR
                                 ( Stack is empty )

                If  ( Stack is NOT empty )

                    {
                        - The node on the end of the outward edge becomes
                        the new growth node.
                        - Link the new growth node to the contouring tree.

                    }  # Endif Stack is not empty

            }

            Until ( Stack is empty )

            - Put the root node of the tree constructed onto the tree head
            pointer list.

        }  # Endif    In_Degree(i,i)=0

    }  # Endfor    For i = 1 to MAX_SIZE


End   # Procedure
```

Figure 2.13
Pseudocode for Constructing the Contouring Tree from the In-degree Matrix

with a test for nodes whose in-degree is zero. When the algorithm finds such a node, it determines its situation number. The algorithm then places the edges corresponding to that situation onto a stack in clockwise, or reverse numerical, order (see *Figure* 2.12).

Once the situation's edges are on the stack, the algorithm then takes one edge at a time from the stack and determines if the edge is directed away from the growth node. If the edge is directed away, the node on the other end of the edge becomes the new growth node. The new growth node is linked to the tree. If the edge is not directed away from the growth node, i.e. it is directed towards the growth node, that edge is discarded. The next edge on the stack is then examined in a similar manner. This continues until either an edge directed away is found, or until the stack is emptied. If a directed away edge is found, a new growth node is established and the process repeats (see *Figure* 2.13). This process continues until all diagonal entries of the in-degree matrix have been examined, and all edges have been added to the contouring trees.

The above section outlines the main steps of the contouring tree growth process. The following sections explain each of those steps in detail.

a. Procedure Search Path

When the contouring algorithm finds a root node in the in-degree matrix (as shown at the top of *Figure* 2.13), procedure **SearchPath** is called (see *Figure* 2.14). This procedure is the one that searches through all paths from the root node to the other nodes of the directed graph. During this search, the procedure links to the contouring tree any previously unvisited nodes that pass the growth node eligibility test. The procedure finishes when all reachable nodes have been visited. A node is reachable if it can be visited from the root by way of a directed path through previously unvisited nodes.

During the first call of procedure SearchPath, a node corresponding to the root node of the contouring tree is created. Procedure **EdgesInSitu<Number>** is then called (see *Figure* 2.15). The procedure called is determined by the situation number of the root node. Procedure EdgesInSitu<Number> has the list of edges corresponding to the situation's number. These edges are processed by procedure EdgesInSitu<Number> by way

Some descriptions

NODE(RowNum): the node defined by "RowNum" in the in-degree matrix
NODE(Coord): the node defined by "Coord" in the in-degree matrix
Deter: takes the zero value (0) for the root node otherwise. the one value (1)
EdgeNum: the edge number used to visit the growth node
ExistWay: true for the outward edge, otherwise false
FirstCall: true if the procedure is in first call. otherwise false
LastEdge: true if the last edge on the edge list is being checked. otherwise false

```
Procedure SearchPath (Situation.Coord.RowNum.EdgeNum)

    # If there is a path from the node defined by "RowNum" to the node
    # defined by NODE(Coord) in the in-degree matrix.

    If (In-Degree(RowNum,NODE(Coord)) == -1 )

    {
        # Reset the growth node to be NODE(Coord).

        RowNum <-- NODE(Coord)
        ExistWay <-- True
    }

    Else ExistWay <-- False

    # If the growth node has already been visited and the edge is
    # directed away from the growth node.

    If (( NODE(RowNum) has already been visited ) AND ( ExistWay ))

    {
        - Link the NODE(RowNum) to the contouring tree

        - Link the one or two edge(s) immediately adjacent to the edge
        defined by EdgeNum (see Figure 2.17)
    }

    Else

    {
        - Link the new growth node to the contouring tree. depending upon
        the values of "FirstCall". "LastEdge" and "ExistWay" (see Figure
        2.16)

        If ( ExistWay ) OR ( FirstCall )

        {
            - Issue that the node defined by NODE(Coord) has been visited

            - Assign zero value to "Deter" if the growth node is the root
            node. otherwise one value (1) is assigned.

            - Find the edge number of the edge used to visit the growth node
            and then assign that number to "EdgeNum"
```

Figure 2.14

Pseudocode Of Procedure Search Path

```
# Push all the edges on the edge list of situation 1 onto a stack
# by calling procedure EdgesInSitu_1

If ( Situation == 1 )
    EdgesInSitu_1(Deter,EdgeNum,Coord,RowNum)

# Push all the edges on the edge list of situation 2 onto a stack
# by calling procedure EdgesInSitu_2

Else if ( Situation == 2 )
    EdgesInSitu_2(Deter,EdgeNum,Coord,RowNum)

# Push all the edges on the edge list of situation 3 onto a stack
# by calling procedure EdgesInSitu_3

Else if ( Situation == 3 )
    EdgesInSitu_3(Deter,EdgeNum,Coord,RowNum)

..........
..........
..... ....

# Push all the edges on the edge list of situation 9 onto a stack
# by calling procedure EdgesInSitu_9

Else if ( Situation == 9 )
    EdgesInSitu_9(Deter,EdgeNum,Coord,RowNum)

# Push all the edges on the edge list of situation 10 onto a stack
# by calling procedure EdgesInSitu_10

Else EdgesInSitu_10(Deter,EdgeNum,Coord,RowNum)

    } # if (ExistWay) OR ....
} # Endif Else

End # procedure
```

Figure 2.14 (continued)

Pseudocode Of Procedure Search Path

```
# Sample code for situation 1.
Procedure EdgesInSitu_1 (Deter,EdgeNum,X,Y,RowNum)
    # Push all the edges on the edge list of situation 1 onto a stack.
    For i = 1 to ( Deter + 2 )
    # "Deter" takes value one (1) if the growth node is not the root node,
    # otherwise it takes zero value.
        # Put the edges on the edge list of situation 1
        # in a counterclockwise order.
        If ( EdgeNum == 3 )
            EdgeNum <-- 1
        Else  EdgeNum <-- EdgeNum + 1
        # If the last edge on the edge list of situation 1 is checked.
        if ( i == (Deter + 2) )
            LastEdge <-- true
        # Check edge number 1 of situation 1.
        If ( EdgeNum == 1 )
        {
            # If the maximum value of X coordinate is two, then edge number 1
            goes to edge number 3 of situation 3, otherwise it goes to edge
            number 2 of situation 5.
            If ( (XMAX == 2 )
                SearchPath( 3, X+1, Y, NODE(X,Y), 3)
            Else SearchPath( 2, X+1, Y, NODE(X,Y), 5)
        } # end if ( EdgeNum == 1 )
        # Check edge number 2 of situation 1.
        If ( EdgeNum == 2 )
            # Edge number 2 of situation 1 always goes to edge number 1 of
            # situation 4.
            SearchPath( 4, XMAX+X, Y, NODE(X,Y), 1)
```

Figure 2.15

Pseudocode of the Algorithm of the Tree Growth Process

```
        # Check edge number 3 of situation 1.

        If ( EdgeNum == 3 )
        {

            # If the maximum value of Y coordinate is two, then edge number 3
            # of situation 1 goes to edge number 1 of situation 8, otherwise
            # it goes to edge number 1 of situation 5 (see Figure 2.12).

            If ( YMAX == 2 )
                SearchPath( 8, X, Y+1, NODE(X,Y), 1)

            Else SearchPath( 5, X, Y+1, NODE(X,Y), 1)

        } # end if ( EdgeNum == 3 )
    } # Endfor for i=1 ...
End # Procedure


# General algorithm for all situations

Procedure EdgesInSitu_<Num> (Deter,EdgeNum,Coord,RowNum)

    # MAXEDGE is the maximum number of edges belonging to the node
    # corresponding to situation <Num>

    # Push all the edges of situation <Number> onto a stack.

    For i = 1 to (Deter + (MAXEDGE-1)
    {

        # Put the edges on the list into a counterclockwise order

        If ( EdgeNum == MAXEDGE )
            EdgeNum <-- 1

        Else EdgeNum <-- EdgeNum + 1

        # Take one edge from the top of the stack and then check it.

        # if the edge number taken from the stack is one.

        If (EdgeNum == 1)
        {

            - Find which situation edge number 1 must be connected to and then
            assign that situation number to "Situation"

            - Assign the new value to "Coord" for the coordinate of the growth
            node

            - Find the edge number of the edge used to visit the growth node
            and then assign that edge number to "EdgeNum".

            - Assign the new value to "RowNum" to show the node in the in-
            degree matrix associated with the previous growth node in the grid
```

Figure 2.15 (continued)

Pseudocode of the Algorithm of the Tree Growth Process

```
        # Call procedure SearchPath for finding all the paths from edge
        number 1 of situation <Number> to other nodes in the grid.

        SearchPath (Situation,Coord,RowNum,EdgeNum)

    } # end if (EdgeNum == 1)

    # if the edge number taken from the stack is two.

    Else If (EdgeNum == 2)
    {
        - Same above process is repeated for edge number 2

    }

    # if the edge number taken from the stack is three.

    Else if (EdgeNum == 3)
    {
        - Same above process is repeated for edge number 3
    }

    ..............
    ..............
    ..............
    ..............

    # if the edge number taken from the stack is MAXEDGE.

    Else if (EdgeNum == MAXEDGE)
    {
        - Same above process is repeated for edge number MAXEDGE

    }

  } # end for i = 1

End # Procedure
```

Figure 2.15 (continued)

Pseudocode of the Algorithm of the Tree Growth Process

of calls to procedure SearchPath. SearchPath determines each edge's direction with respect to the growth node. If the edge is directed away from the growth node, then the node on the end of that edge becomes the new growth node.

In the above, we note that procedures SearchPath and EdgesInSitu<Number> call each other recursively. Procedure EdgesInSitu<Number> pushes the ordered edges onto a stack. Procedure SearchPath takes an edge from the top of that stack and then checks if that edge is directed away from the growth node. This process repeats until the stack is empty of edges. At that point the contouring tree gro' h process is complete.

The procedure used to link a new growth node to the contouring tree, **GrowingTree**, is shown in *Figure* 2.16. The notation we use in that procedure is quite standard. **CHILD(NODE)** represents the child pointer of the node. **SIBLING(NODE)** the sibling pointer of the node and **PRED(NODE)** the predecessor node in the contouring tree.

Procedure SearchPath always calls procedure GrowingTree. Growing ee is the procedure that links new growth nodes to the contouring tree. In the first call, GrowingTree creates a node with the necessary data and then assigns this node to be the root node of the tree. The next growth nodes are linked to the field CHILD(NODE) of the root node. Each new growth node is linked to the field CHILD(NODE) of the previous growth node. This continues until the last edge on the list associated with the growth node has been traversed and there are no further nodes reachable from the growth node. When this condition occurs, procedure GrowingTree resets the growth node to be the closest to the right sibling node in the tree. The next growth node is linked to the field SIBLING(NODE) of that node. The next growth node is linked to the field CHILD(NODE) of the newly linked node. This tree growth process is repeated until all the edges belonging to the root node in the directed graph are exhausted.

If procedure SearchPath runs into a node which has already been visited during the contouring tree growth process, then SearchPath calls procedure **SharedEdge**. (see *Figure* 2.17). SharedEdge processes the edges immediately adjacent to the edge used to visit the growth node. If the adjacent edges are directed away from the growth node, then the nodes on the ends of the

```
Procedure GrowingTree (Coord.ExistWay.LastEdge)

    If ( FirstCall )
        {

            - Create the new growth node for the tree
            - Put the necessary data on this node
            - Assign this node to be the root node of the tree
            - Issue that the next growth node will be linked to field
            CHILD(NODE)

        }
    Else If ( The edge is directed away from the growth node )
            AND
            ( LastEdge )
    {

            - Link the growth node to the tree
            - Reset the growth node to be the closest to the right sibling node in
            the tree.

    }
    Else If ( ExistWay )
    {

        # Growth node was moved over to the closest right sibling node.

        If ( Declared field to be linked is SIBLING(NODE) )
        {

                - Link the growth node to the field SIBLING(NODE)
                - Issue that the next growth node will be linked to the field
                CHILD(NODE)

        }
        Else
        {

                - Continue linking the new growth node to the same previous
                field (It can be the field CHILD(NODE) or SIBLING(NODE)).

        }
    }
    Else If (( The edge is NOT directed away from the growth node )
            AND
            ( LastEdge ))
    {

            - Find the immediate father node in the tree.
            - Issue that the next growth node will be linked to the field
            SIBLING(NODE)

    }

End    # GrowingTree
```

Figure 2.16

Pseudocode Of Algorithm Linking The Growth Nodes To The Contouring Tree outward edges are linked to the tree, as children of the growth node. Edges directed towards the growth node are ignored.

When procedure SearchPath reaches the empty stack condition, the contouring tree growth process is over. This same process is repeated for all root nodes in the directed graph. *Figure* 2.18 illustrates the two contouring trees created from the in-degree matrix of *Figure* 2.6.

3. Drawing Command Placement

Drawing commands are placed in the contouring tree to indicate when a line enters the region represented by the contouring tree either from a neighboring subgrid or from a location off of the grid. If we look at the structure of the contouring tree and consider that during the traversal, the edges are examined in a counterclockwise, and downward ordering from the root, we note that we need to place setpoint drawing commands on the lower valued node of each edge that presents a new lowest value for the tree. (Note that the drawing command **Setpoint** indicates to the display device that it should move its "drawing instrument", i.e. electron beam, pen, etc., in a non-drawing mode to the specified location, and that it should then place that drawing instrument into a drawing mode. **Drawto** indicates to the display device that it should move its drawing instrument in a drawing mode to the specified location. **Drawpoint** indicates to the display device that it should move its drawing instrument in a non-drawing mode to the specified location, and that it should then turn that drawing instrument on for the space of a single point.) We insert these drawing commands by way of a pre-order traversal of the directed tree, placing a setpoint command on each node that is a new lowest value for the tree. This drawing command placement strategy is based upon the fact that if we have a contour level for which we desire a picture, the first drawing command we generate for any contouring tree is a setpoint. Although fairly effective, this procedure does not provide a complete solution to drawing command insertion.

C. DRAWING COMMAND PLACEMENT PROBLEMS

1. Split Edge Problem

The drawing command placement strategy outlined above does not provide a complete solution to drawing command insertion. Some neighboring

```
Procedure SharedNode (Situation.Coord.RowNum.EdgeNum)

    If ( Situation == 1 )
            # If we are looking at edge number 1 of situation 1.

            If ( EdgeNum == 1 )
                {
                    - Find the coordinates of the edges adjacent to edge number
                      1 in situation 1 (see Figure 2.12).
                }
            # If we are looking at edge number 2 of situation 1.

            Else If ( EdgeNum == 2 )
                {
                    - Find the coordinates of the edges adjacent to edge number
                      2 in situation 1 (see Figure 2.12).
                }
            Else ( Do the same above process for EdgeNum = 3 )
    Else If ( Situation == 2 )
            # If we are looking at edge number 1 of situation 2.

            If ( EdgeNum == 1 )
                {
                    - Find the coordinates of the edges adjacent to edge number
                      1 in situation 2 (see Figure 2.12).
                }
            # If we are looking at edge number 2 of situation 2.

            Else If ( EdgeNum == 2 )
                {
                    - Find the coordinates of the edges adjacent to edge number
                      2 in situation 2 (see Figure 2.12).
                }
            ................
            ................
            Else If ( EdgeNum == 5 )
                {
                    - Find the coordinates of the edges adjacent to edge number
                      5 in situation 2 (see Figure 2.12).
                }
    Else If ( Situation == 3 )
            {
                - Find the coordinates of the edges adjacent to the edge number
                  defined by "EdgeNum" in situation 3 (as above).
            }
    ................
    ................
    Else If ( Situation == 10 )
            {
                ................................
            }
    # Check whether the adjacent edges are directed away from the growth node.

    If ( The adjacent edge(s) are directed away from the growth node )
            {
                - Link the adjacent edge(s) to the contouring tree
            {
End   # procedure SharedNode
```

Figure 2.17
Pseudocode Of Algorithm Evaluating Already Visited Nodes

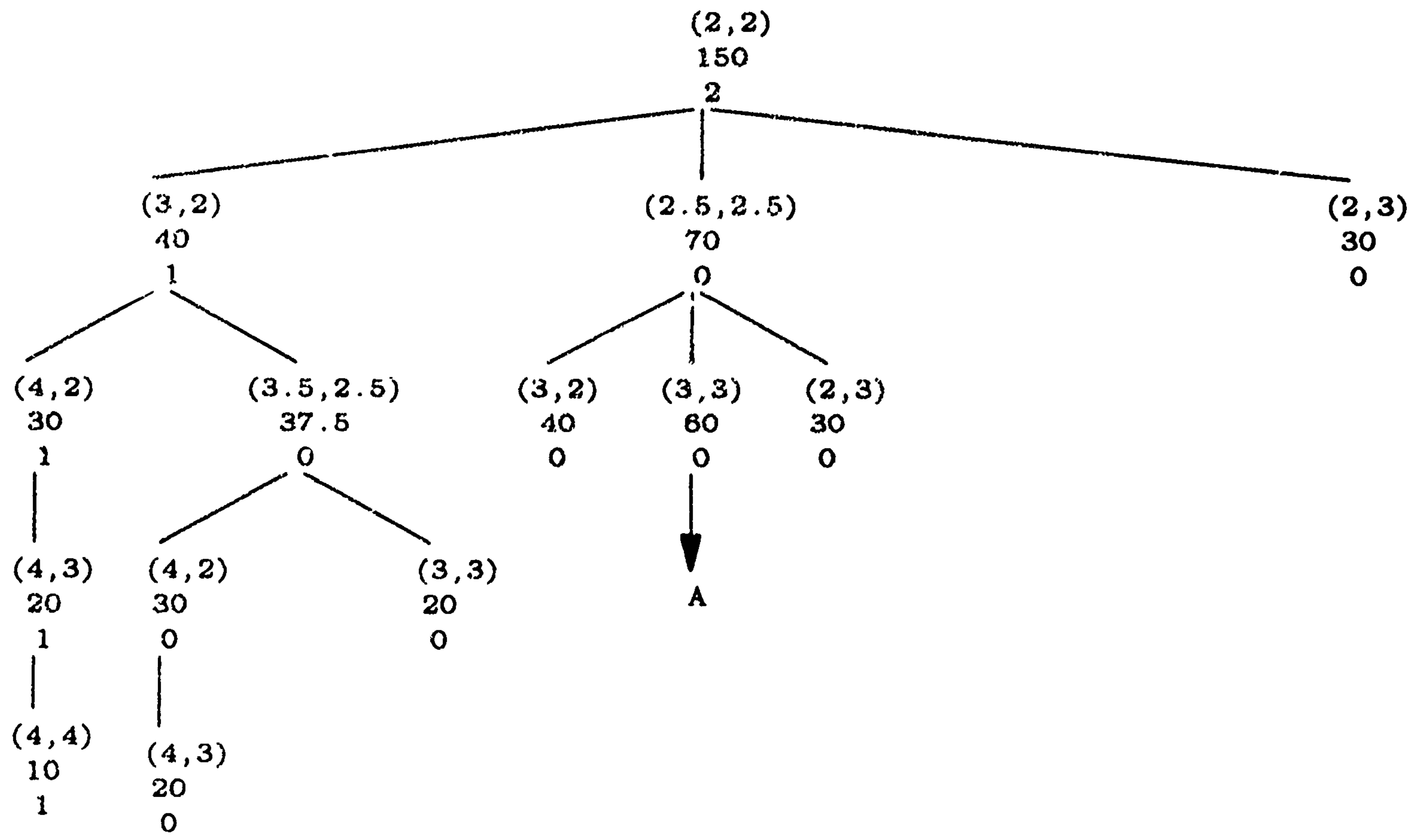


Figure 2.18a
Representation Of The First Contouring Tree Rooted At Node (2,2).
Tree is Created From The In-Degree Matrix Of Figure 2.6

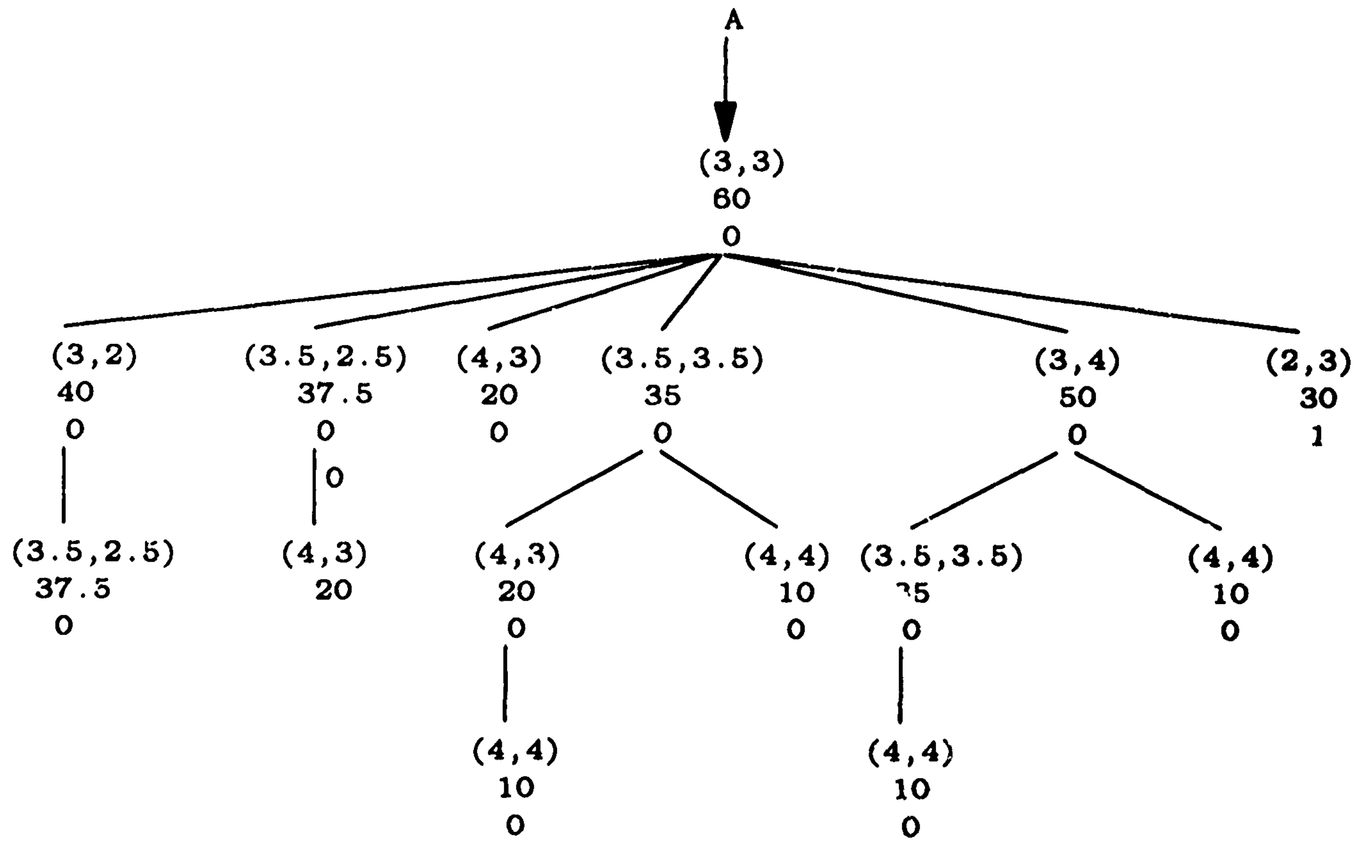


Figure 2.18a (continued)
Representation Of The First Contouring Tree Rooted At Node (2,2).
Tree is Created From The In-Degree Matrix Of Figure 2.6

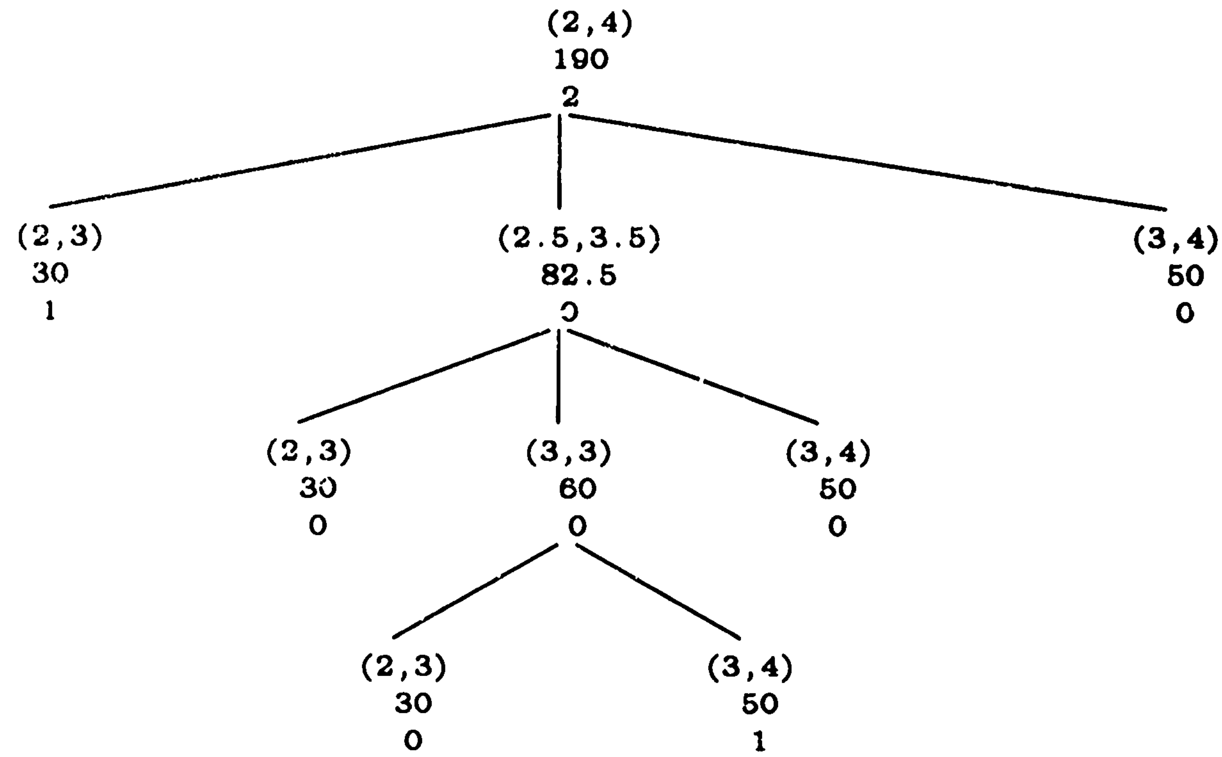


Figure 2.18b
Representation Of The First Contouring Tree Rooted At Node (2,4).
Tree is Created From The In-Degree Matrix Of Figure 2.6

edges in the contouring tree, i.e. edges sharing an ancestor node, have a "split" between them, i.e. the edges are not immediate counterclockwise neighbors in the original grid. In this case, we must indicate the discontinuity in the contouring tree. We register the discontinuity on the lower valued node of the edge where the discontinuity occurs. For example, in *Figure* 1.3a the edges (3,3)-(3,2) and (3,3)-(2,3) are neighbors in the contouring tree but are not immediate neighbors in the original grid. We indicate this split by placing a "1" on the lower valued node of edge (3,3)-(2,3).

In order to recognize the nodes that require a drawing command indicating a split edge in the contouring tree, we must take care of all possible places where split edges occur in the larger grid. The algorithm of *Figure* 2.19 solves this problem.

The main idea behind this algorithm comes from the definition of the split edge problem. During drawing command placement, the father node pointer and his child node pointer are given to procedure **Split_Edge_Control**. This procedure determines to which situation the father node belongs. The edge number between the father node and the child node is then found. When the edge number is known, the edge immediately adjacent to that edge in the grid is easily determined. For the continuous case, i.e. non-split edge case, this edge also exists in the contouring tree. The algorithm finds the edge number between the father node and the next child node in the contouring tree. The adjacent edge number in the tree is compared with the edge number in the grid. If these two edges are the same, it means that there is no split edge problem. If the edges are different, a split edge problem exists. We put a setpoint indicator on the lower valued node of the split edge. This procedure also takes care of edges lacking an adjacent edge in counterclockwise order. For example, edge number 3 in situation 1 has no adjacent edge in counterclockwise order. In this case, if there is another child node in the contouring tree, a split edge condition exists.

2. Edge Duplication Problem

The core problems with the 2x2 subgrid algorithm all concern issues of picture efficiency. Since the display generated for each 2x2 subgrid is generated independently of any neighboring 2x2 subgrids, equivalued lines at the contour

```
Procedure Split_Edge_Control (FATHER(NODE).CHILD/SIBLING(NODE))
  # "/" means "OR"

    # Find the edge number of the edge in the grid and the edge number in the
    tree construction.

    - Find to which situation FATHER(NODE) belongs.

    - Find the edge number between FATHER(NODE) and
    CHILD/SIBLING(NODE) in the tree construction.

    - Find the edge number immediately adjacent to that edge number in the
    grid.

    - Find the next SIBLING(NODE) -next child of the same father-

    - Find the edge number between FATHER(NODE) and SIBLING(NODE)
    in the tree construction.

    # Determine if there is a split edge.

    If (The adjacent edge number in the tree construction is NOT the same
        as the adjacent edge number in the grid )   # Split Edge Exists

    {

        - Put a Setpoint command in node.

    }

    Else        # No Split Edge

    {

        - Put a Drawto command in node.

    }

End     # Split_Edge_Control
```

Figure 2.19
Pseudocode Of Algorithm Solving The Split Edge Problem

level on the border of a 2x2 subgrid are duplicated. A similar problem occurs for subgrid corner values that equal the contour level. If we display either of the above cases on a calligraphic display device, we see a bright line for the equivalued edge, and a bright point for the grid value equal to the contour level. Another problem, also due to the independent computation of each 2x2 subgrid is that no ordering is provided for coordinates that come out of this algorithm. These are the problems with the 2x2 subgrid algorithm.

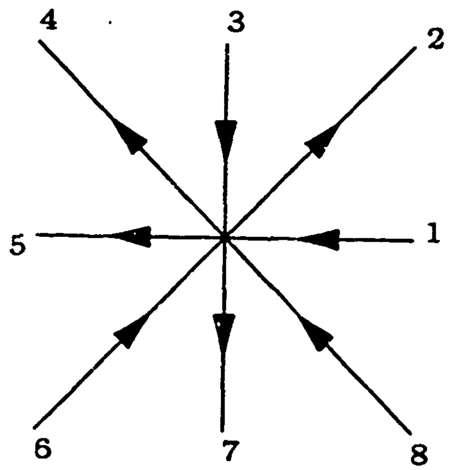
In the Large Contouring Tree Algorithm, we eliminate contour line duplications in two ways. First, during the tree growth process we don't allow repeated subtrees to be linked to the contouring tree. Repeated subtrees cause the duplication of contour lines during the traversal process. For this reason, when we visit any node more than once during the tree growth process. we take into account only edges immediately adjacent to the edge used to visit the growth node. If these adjacent edges are directed away from the growth node, then we link those adjacent edges to the tree. Edges directed towards the growth node are ignored. *Figure* 2.20 shows how to handle a node that has been visited more than once.

The second procedure we use to eliminate contour line duplications is to keep track of the coordinates of equivalued edges at the contour level in order to prevent the equivalued edges from appearing more than once. Before outputing the coordinate and drawing command of an equivalued edge. we check to determine if there is another equivalued edge with the same coordinates as the one already on hand. If there is. we discard that second set of coordinates and drawing instructions.

3. Decision On Closed Contour Lines

The contours at level 100 on *Figure* 1.3 are closed contours. forming simple. connected loops. The contours at level 50 on *Figure* 1.2 are open contours. In the creation of the closed contours. we can't complete the connected loops by traversing contouring tree. The contour lines between the starting point and the ending point stay open. We need a procedure that determines when contour lines should be closed.

ASSUMPTION: the growth node is at the center of this picture. The node is a situation 6 node.



CASE 1: the first visit is via edge number 6
Edge numbers to be linked are 7,2,4 and 5
CASE 2: the second visit is via edge number 8
Edge number to be linked is 7
CASE 3: the third visit is via edge number 1
Edge number to be linked is 2
CASE 4: the fourth visit is via edge number 3
Edge numbers to be linked are 4 and 2

Figure 2.20
Elimination Of Repeated Subtree During The Tree Growth Process

Two conditions must occur for the closed contours decision. First, the root node of the contouring tree must belong to situation 6. Situation 6 is the only situation with a complete set of adjacent edges that is eligible to serve as the root of a contouring tree. The second condition is that the starting and the ending coordinates of the concerned contour cannot be on the border lines of the grid. If the coordinates are on the border lines (outermost lines) of the grid, then we don't close the contour.

D. DISPLAY GENERATION

Display generation from a contouring tree is accomplished by performing a pre-order traversal of that contouring tree, producing a coordinate and drawing instruction whenever the desired contour level is found to be within the range of an edge of the contouring tree. A pre-order traversal visits the root, the left subtree (CHILD(NODE)) and then the next right subtree, etc. An edge's range is defined to be the set of values between those associated with the nodes on either end of edge. More precisely, we say a contour level is within an edge if the following condition holds:

lower_node's_value <= contour_level < higher_node's_value

The drawing instruction issued for each edge is the one associated with the lower valued node of the edge. The coordinate for each of these edges is generated by a linear interpolation of the edge's end point coordinates according to the decrease in contour level along the edge.

There are some subtleties not evident from the above that are best detailed using a pseudocode description of the traversal algorithm. *Figure* 2.21 depicts the traversal procedure for the contouring tree assuming a particular data organization. The pointers to the descendent nodes of NODE are CHILD(NODE) and SIBLING(NODE). For each node of the contouring tree, there are three pieces of information: the value associated with the node, VALUE(NODE), the coordinate associated with the node, XYZ(NODE), and the connectivity with the node, CONN(NODE).

The generation of coordinates and drawing instructions from a contouring tree begins with routine CONTOUR_SUBGRID of *Figure* 2.21. That routine receives a pointer to the root node of the contouring tree. It then starts the traversal by calling routine VISIT with that root node. Routine VISIT checks to see if the edge defined by the passed in node and that node's ancestor, NODE and ANCESTOR, contains the contour level. If the edge does contain the contour level, the edge intersection coordinate is computed using linear interpolation and issued to the display along with the connectivity associated with that node.

CONN(NODE). If we issue a coordinate and connectivity for a node, we need to check the subtree under that node for equivalued edges. If an equivalued edge at the contour level is found, a coordinate and drawing instruction pair are issued for that edge (routine VISIT_SUBTREE). Once a coordinate and drawing instruction pair have been issued for an edge, and once the subtree beneath that edge has been investigated for equivalued edges, further traversal of that subtree is terminated. If an edge is found not to contain the contour level, the traversal continues as depicted at the bottom of routine VISIT.

The pre-order traversal procedure described above generates the coordinates and drawing instructions for the part of the grid the contouring tree represents. *Figure* 2.22 shows the coordinates generated for the 3x3 grid of *Figure* 2.1 *Figure* 2.2 shows the contour lines drawn by using those coordinates and drawing commands.

using the situation naming scheme described in chapter 2. The procedure works in the following fashion.

The first step is to determine to which situation each node of the directed graph belongs. The second step is to examine the edges belonging to the situation and to set the appropriate values in the in-degree matrix. For example, for node (1,1), we call procedure situation_1. In this procedure, the directions of three edges are examined and recorded in the in-degree matrix. The first edge examined is (1,1)-(2,1). The value of minus one is put in D(2,1) to indicate the edge is directed from node 2 to node 1. For the second edge, (1,1)-(1.5,1.5), the value of minus one is put in D(5,1) to indicate the edge is directed from node 5 to node 1. For the third edge, (1,1)-(1,2), the value of minus one is put in D(4,1). Once the edges associated with node (1,1) have been examined, the in-degree matrix construction procedure then performs a similar set of operations on node (2,1).

Once the above operations have been performed on all nodes of the directed graph, the in-degree matrix is completed by computing the values on its diagonal. The value of in-degree matrix D(i,i) indicates the number of edges directed towards node i in the directed graph. This value is computed by summing the total number of minus one values in column i. *Figure* 3.5 shows the in-degree matrix for the directed graph of *Figure* 3.4.

B. CONTOURING TREE FOR THE EXAMPLE GRID

Contouring tree construction is a growth process that begins from each node indicated in the in-degree matrix as lacking incoming edges. These nodes, termed root nodes, have D(i,i) values of zero. The in-degree matrix of *Figure* 3.5 has three zero valued entries on its diagonal at node 3, 6, and 19. The contouring tree construction procedure grows a separate contouring tree from each of these nodes.

Figure 3.6 shows the three contouring trees created out of the in-degree matrix. Let's try to create the contouring tree rooted at node (2,2).

There are eight edges connected to node (2,2). We push those edges into a stack in a clockwise order. The starting edge for this order is selected by the rule shown on *Figure* 2.12. In our case, edge (2,2)-(2.5,1.5) is the starting edge. Edges (2,2)-(2.5,1.5), (2,2)-(2,1), (2,2)-(1.5,1.5), (2,2)-(1,2), (2,2)-(1.5,2.5), (2,2)-(2,3),

```
        # Check subtrees of this node for equivalued edges.
        VISIT_SUBTREE(CHILD(NODE),NODE)
        VISIT_SUBTREE(SIBLING(NODE),NODE)

        return # no need to examine the subtree further.

    } # endif coordinates were generated for an edge.

    VISIT(CHILD(NODE),NODE) # visit left subtree.
    VISIT(SIBLING(NODE),NODE) # visit right subtree.

    return

end

Procedure VISIT_SUBTREE(SUBNODE,SUBANCESTOR)
    if(SUBNODE == NULL)
    {
        return
    }

    if(VALUE(SUBNODE) == CONTOUR_LEVEL)
    {

        Issue coordinates for the equivalued edge.
        Setpoint on XYZ(SUBANCESTOR).
        Drawto XYZ(SUBNODE).

    }

    VISIT_SUBTREE(CHILD(SUBNODE),SUBNODE)
    VISIT_SUBTREE(SIBLING(SUBNODE),SUBNODE)

    return

end
```

Figure 2.21 (continued)

Pseudocode of the Traversal Algorithm for the Contouring Tree

First Tree Rooted At Value 150.00			
Level 50			
X	Y	Z	D
2.9091	2.0000	0.0000	1
2.8333	2.1667	0.0000	0
3.0000	2.5000	0.0000	0
3.2222	2.7778	0.0000	0
3.2500	3.0000	0.0000	0
3.2000	3.2000	0.0000	0
3.0000	4.0000	0.0000	0
2.6667	3.0000	0.0000	1
2.2500	2.7500	0.0000	0
2.0000	2.8333	0.0000	0

Second Tree Rooted At Value 190.00			
Level 50			
X	Y	Z	D
2.0000	3.1250	0.0000	1
2.1905	3.1905	0.0000	0
2.6667	3.0000	0.0000	0
3.0000	4.0000	0.0000	1
3.0000	4.0000	0.0000	0

First Tree Rooted At Value 150.00			
Level 100			
X	Y	Z	D
2.4545	2.0000	0.0000	1
2.3125	2.3125	0.0000	0
2.0000	2.4167	0.0000	0

Second Tree Rooted At Value 190.00			
Level 100			
X	Y	Z	D
2.0000	3.4375	0.0000	1
2.4186	3.5814	0.0000	0
2.6429	4.0000	0.0000	0

Column D is the drawing command, i.e. 1 - SETPOINT 0 - DRAWTO

Figure 2.22

Coordinates Generated For The 3X3 Subgrid With Saddle Point

III. A COMPLETE EXAMPLE

In this chapter we apply the contouring tree algorithm to the 4x5 grid of *Figure* 3.1. We create the in-degree matrix for the grid. We then build the contouring trees from the in-degree matrix. The final step we show is the generation of coordinates and drawing instructions from the contouring trees.

Figure 3.1 shows the density values of the two-dimensional grid of our example. The density values are input to the algorithm as a two-dimensional array. The first step of the algorithm is to compute the average density values for the center points of each 2x2 subgrid of the 4x5 grid. *Figure* 3.2 shows the 4x5 grid with the calculated center point densities.

The second step in the algorithm is to create the directed graph from the 4x5 grid. To get the directed graph from the grid, we assign a direction to each edge of the 4x5 grid using the density value assigned to each node. For the equivalued edge, we assign a direction to the edge that is counterclockwise with respect to the growth node. *Figure* 3.4 shows the directed graph of the 4x5 grid of *Figure* 3.2. In this directed graph, there are no equivalued edges. The arrowheads on the edges of the directed graph point in the direction of the lower valued node. For example, the arrowhead on edge (1,1)-(2,1) is toward to node (1,1).

A. IN-DEGREE MATRIX CREATION

We create the in-degree matrix to reflect the directed graph better. To create the in-degree matrix, we need to provide a mapping from the nodes of the 4x5 grid to the nodes of the in-degree matrix. *Figure* 3.3 shows this mapping. The number on the node in the 4x5 grid is the node number in the in-degree matrix. For example, node (1,1) in the grid is associated with node number 1 in the in-degree matrix. Node (1,2) is associated with node number 2 in the in-degree matrix. Node (3,1) is associated with node number 6, etc.

The mapping procedure that associates a node number in the directed graph with a node number in the in-degree matrix is performed on a case-by-case basis

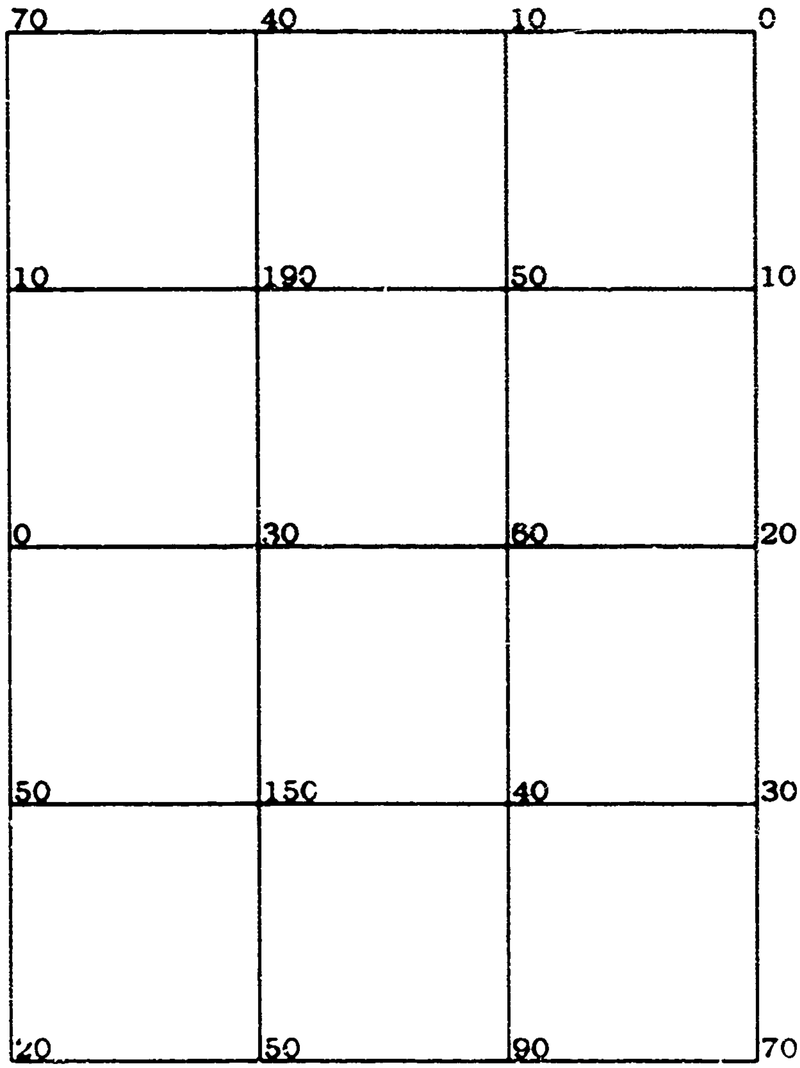


Figure 3.1

An Example 4X5 Grid With Density Values

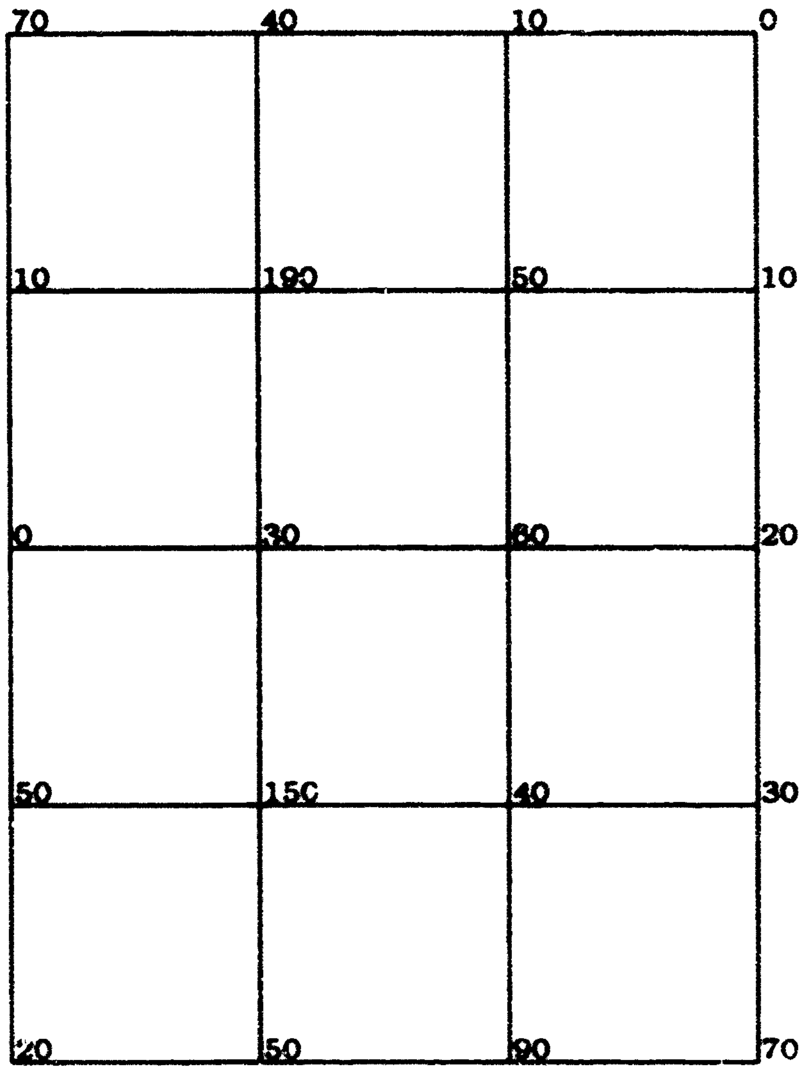


Figure 3.1

An Example 4X5 Grid With Density Values

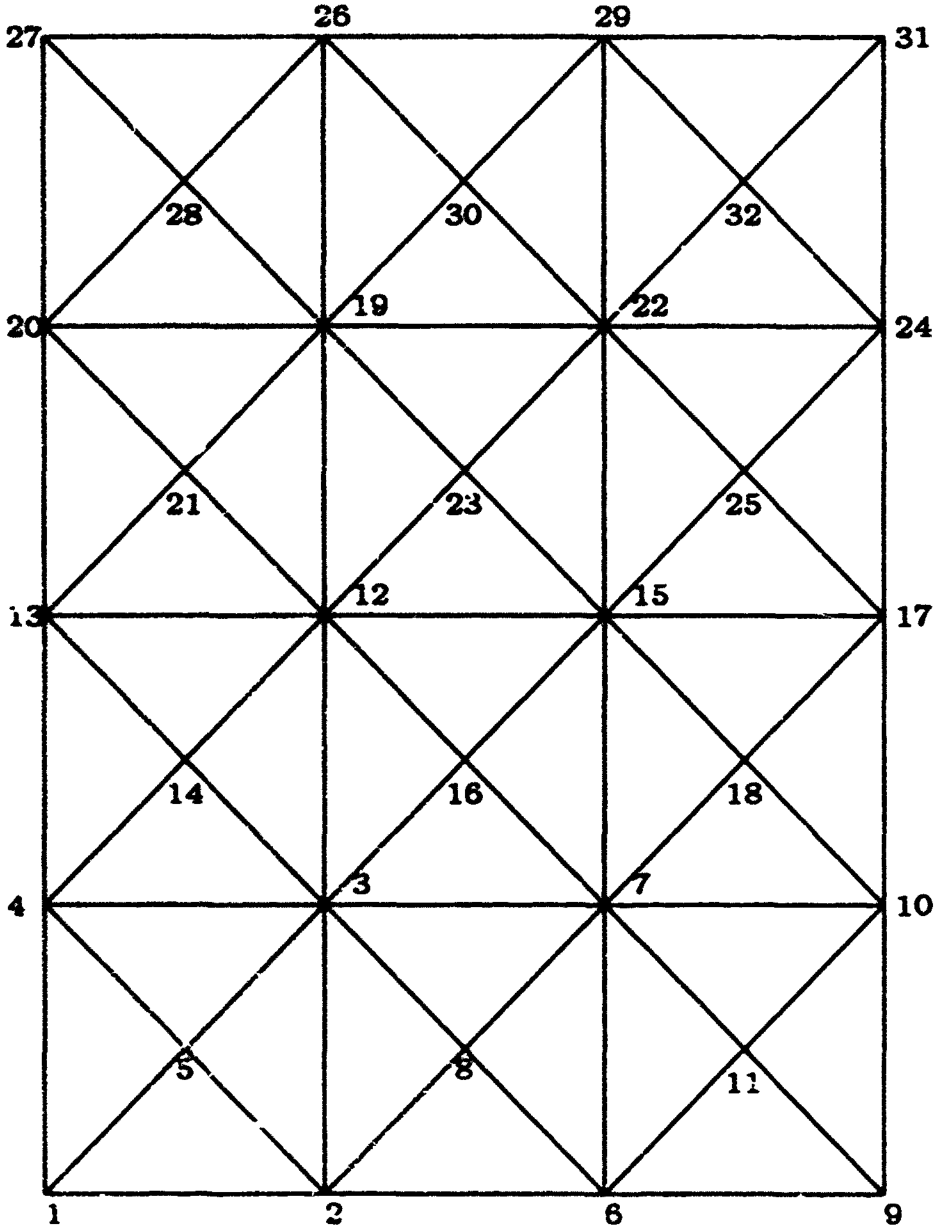


Figure 3.3
The Number On each Node In The Grid Is The Node Number Used For Reference To The In-Degree Matrix

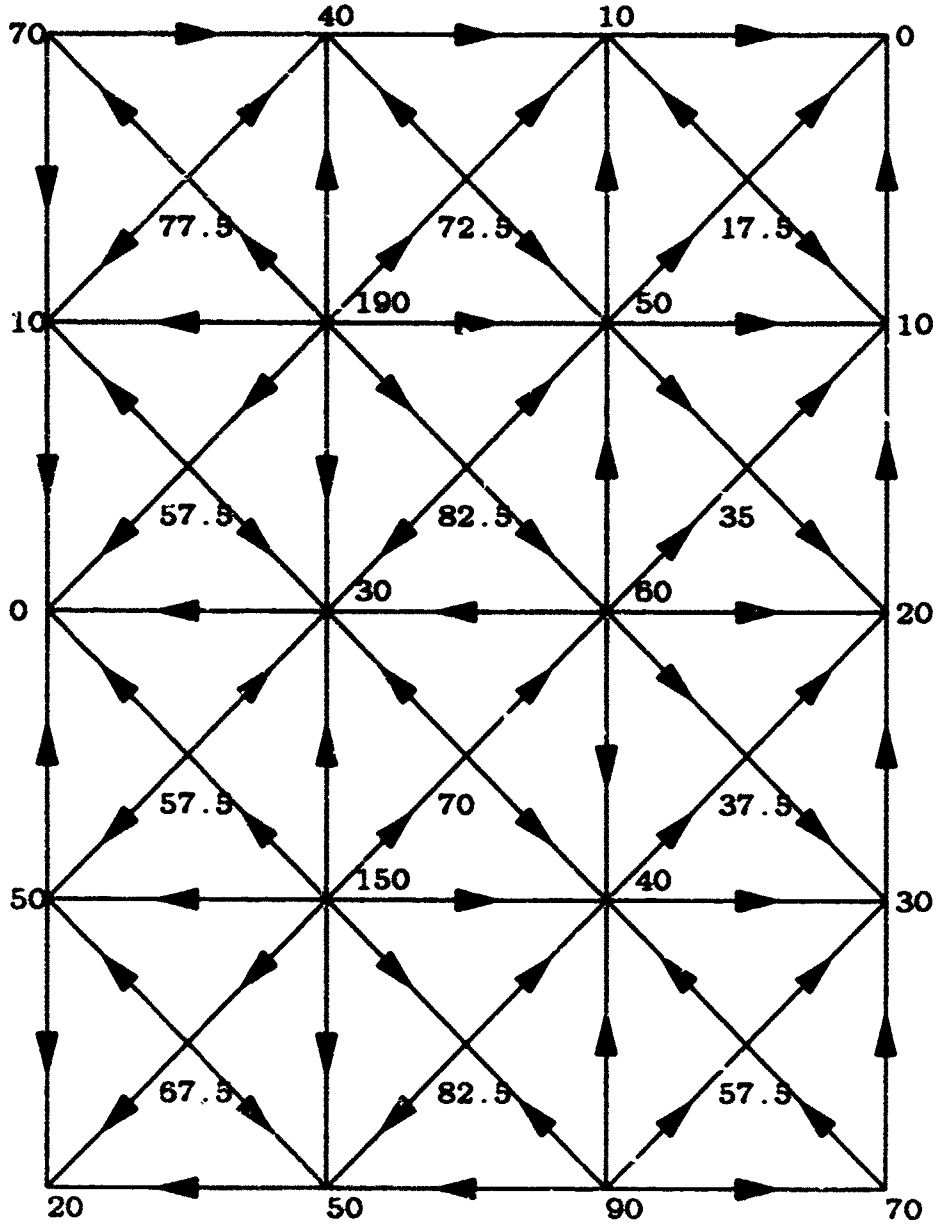


Figure 3.4

Directed Graph For The 4X5 Grid Of Figure 3.2

using the situation naming scheme described in chapter 2. The procedure works in the following fashion.

The first step is to determine to which situation each node of the directed graph belongs. The second step is to examine the edges belonging to the situation and to set the appropriate values in the in-degree matrix. For example, for node (1,1), we call procedure situation_1. In this procedure, the directions of three edges are examined and recorded in the in-degree matrix. The first edge examined is (1,1)-(2,1). The value of minus one is put in D(2,1) to indicate the edge is directed from node 2 to node 1. For the second edge, (1,1)-(1.5,1.5), the value of minus one is put in D(5,1) to indicate the edge is directed from node 5 to node 1. For the third edge, (1,1)-(1,2), the value of minus one is put in D(4,1). Once the edges associated with node (1,1) have been examined, the in-degree matrix construction procedure then performs a similar set of operations on node (2,1).

Once the above operations have been performed on all nodes of the directed graph, the in-degree matrix is completed by computing the values on its diagonal. The value of in-degree matrix D(i,i) indicates the number of edges directed towards node i in the directed graph. This value is computed by summing the total number of minus one values in column i. *Figure* 3.5 shows the in-degree matrix for the directed graph of *Figure* 3.4.

B. CONTOURING TREE FOR THE EXAMPLE GRID

Contouring tree construction is a growth process that begins from each node indicated in the in-degree matrix as lacking incoming edges. These nodes, termed root nodes, have D(i,i) values of zero. The in-degree matrix of *Figure* 3.5 has three zero valued entries on its diagonal at node 3, 6, and 19. The contouring tree construction procedure grows a separate contouring tree from each of these nodes.

Figure 3.6 shows the three contouring trees created out of the in-degree matrix. Let's try to create the contouring tree rooted at node (2,2).

There are eight edges connected to node (2,2). We push those edges into a stack in a clockwise order. The starting edge for this order is selected by the rule shown on *Figure* 2.12. In our case, edge (2,2)-(2.5,1.5) is the starting edge. Edges (2,2)-(2.5,1.5), (2,2)-(2,1), (2,2)-(1.5,1.5), (2,2)-(1,2), (2,2)-(1.5,2.5), (2,2)-(2,3),

D(i,j)	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
1	3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
2	-1	4	0	0	0	0	0	0	0	0	0	0	0	0	0	0
3	0	-1	0	-1	-1	0	-1	-1	0	0	0	-1	0	-1	0	-1
4	-1	0	0	3	0	0	0	0	0	0	0	0	-1	0	0	0
5	-1	-1	0	-1	1	0	0	0	0	0	0	0	0	0	0	0
6	0	-1	0	0	0	0	-1	-1	-1	0	-1	0	0	0	0	0
7	0	0	0	0	0	0	6	0	0	-1	0	0	0	0	0	0
8	0	-1	0	0	0	0	-1	2	0	0	0	0	0	0	0	0
9	0	0	0	0	0	0	0	0	1	-1	-1	0	0	0	0	0
10	0	0	0	0	0	0	0	0	0	4	0	0	0	0	0	0
11	0	0	0	0	0	0	-1	0	0	-1	2	0	0	0	0	0
12	0	0	0	0	0	0	0	0	0	0	0	7	-1	0	0	0
13	0	0	0	0	0	0	0	0	0	0	0	0	5	0	0	0
14	0	0	0	-1	0	0	0	0	0	0	0	-1	-1	1	0	0
15	0	0	0	0	0	0	-1	0	0	0	0	-1	0	0	2	0
16	0	0	0	0	0	0	-1	0	0	0	0	-1	0	0	-1	1
17	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
18	0	0	0	0	0	0	0	0	0	-1	0	0	0	0	0	0
19	0	0	0	0	0	0	0	0	0	0	0	-1	0	0	0	0
20	0	0	0	0	0	0	0	0	0	0	0	0	-1	0	0	0
21	0	0	0	0	0	0	0	0	0	0	0	-1	-1	0	0	0
22	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
23	0	0	0	0	0	0	0	0	0	0	0	-1	0	0	-1	0
24	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
25	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
26	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
27	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
28	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
29	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
30	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
31	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
32	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Figure 3.5

In-Degree Matrix Created From The Directed Graph Of Figure 3.4

D(i,j)	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32
1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
4	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
5	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
6	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
7	0	-1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
8	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
9	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
10	-1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
11	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
12	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
13	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
14	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
15	-1	-1	0	0	0	-1	0	0	-1	0	0	0	0	0	0	0
16	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
17	4	0	0	0	0	0	0	-1	0	0	0	0	0	0	0	0
18	-1	2	0	0	0	0	0	0	0	0	0	0	0	0	0	0
19	0	0	0	-1	-1	-1	-1	0	0	-1	0	-1	0	-1	0	0
20	0	0	0	4	0	0	0	0	0	0	0	0	0	0	0	0
21	0	0	0	-1	1	0	0	0	0	0	0	0	0	0	0	0
22	0	0	0	0	0	4	0	-1	-1	0	0	0	-1	0	0	-1
23	0	0	0	0	0	-1	1	0	0	0	0	0	0	0	0	0
24	0	0	0	0	0	0	0	4	0	0	0	0	0	0	-1	0
25	-1	0	0	0	0	0	0	-1	2	0	0	0	0	0	0	0
26	0	0	0	0	0	0	0	0	0	4	0	0	-1	0	0	0
27	0	0	0	-1	0	0	0	0	0	-1	1	0	0	0	0	0
28	0	0	0	-1	0	0	0	0	0	-1	-1	1	0	0	0	0
29	0	0	0	0	0	0	0	0	0	0	0	0	4	0	-1	0
30	0	0	0	0	0	-1	0	0	0	-1	0	0	-1	1	0	0
31	0	0	0	0	0	0	0	0	0	0	0	0	0	0	3	0
32	0	0	0	0	0	0	0	-1	0	0	0	0	-1	0	-1	1

Figure 3 5 (Continued)

In-Degree Matrix Created From The Directed Graph Of Figure 3 4

(2,2)-(2.5,2.5), and (2,2)-(3,2) are pushed into the stack respectively. When we take an edge from the stack, the order for checking the edges is counterclockwise. The first edge pushed into the stack is the last edge to be checked.

Edge (2,2)-(3,2) on top of the stack is taken and checked. Since this edge is directed away from the growth node, node (3,2) on the end of that outward edge becomes the new growth node We link this node to the contouring tree. We push the edges connected to node (3,2) into the stack in a clockwise order. There are then two differents groups of edges in the stack, one for node (2,2), one for node (3,2). We take edge (3,2)-(2.5,1.5) from the top of the stack and check it. It is an inward edge. The next two edges, (3,2)-(3,1) and (3,2)-(3.5,1.5), are also inward. We skip those edges and take another edge from the stack. Edge (3,2)-(4,2) on top of the stack is outward. Using this edge, we go to node (4,3) and then link that node to the tree. We push all the edges connected to node (4,3) into the stack. This process continues until we reach the empty stack. During this process, we go through all possible paths from the root node to the other nodes in the directed graph. We link the nodes on the paths, to the contouring tree, except for the following case.

If we come to a node which has already been visited, we apply a different procedure to that node. We don't push all the edges connected to that node onto the stack, except the edge(s) immediately adjacent to the edge used to come to that node. If these edges are outward, then we link the node on the end of the outward edge to the tree. We don't push any edge connected to this newly linked node. In other words, we don't take into account the descendents of that node. We then go back to the previous node. The reason for this is that the outward edges, connected to the already visited node, have been linked before to the contouring tree. If we again link those edges to the tree, then we create repeated subtrees in the contouring tree. Repeated subtrees cause the duplication of the contour lines. Let's see how we apply this rule to the directed graph of *Figure* 3.4.

After creating the first tree rooted at node (2,2), we go from the root node to node (3,2) by using edge (2,2)-(3,2). After going through all the paths from node (3,2) to the other nodes in the directed graph, we go back to the stack and check edge (2,2)-(2.5,2.5) on top of the stack. This edge is outward. Using that edge, we

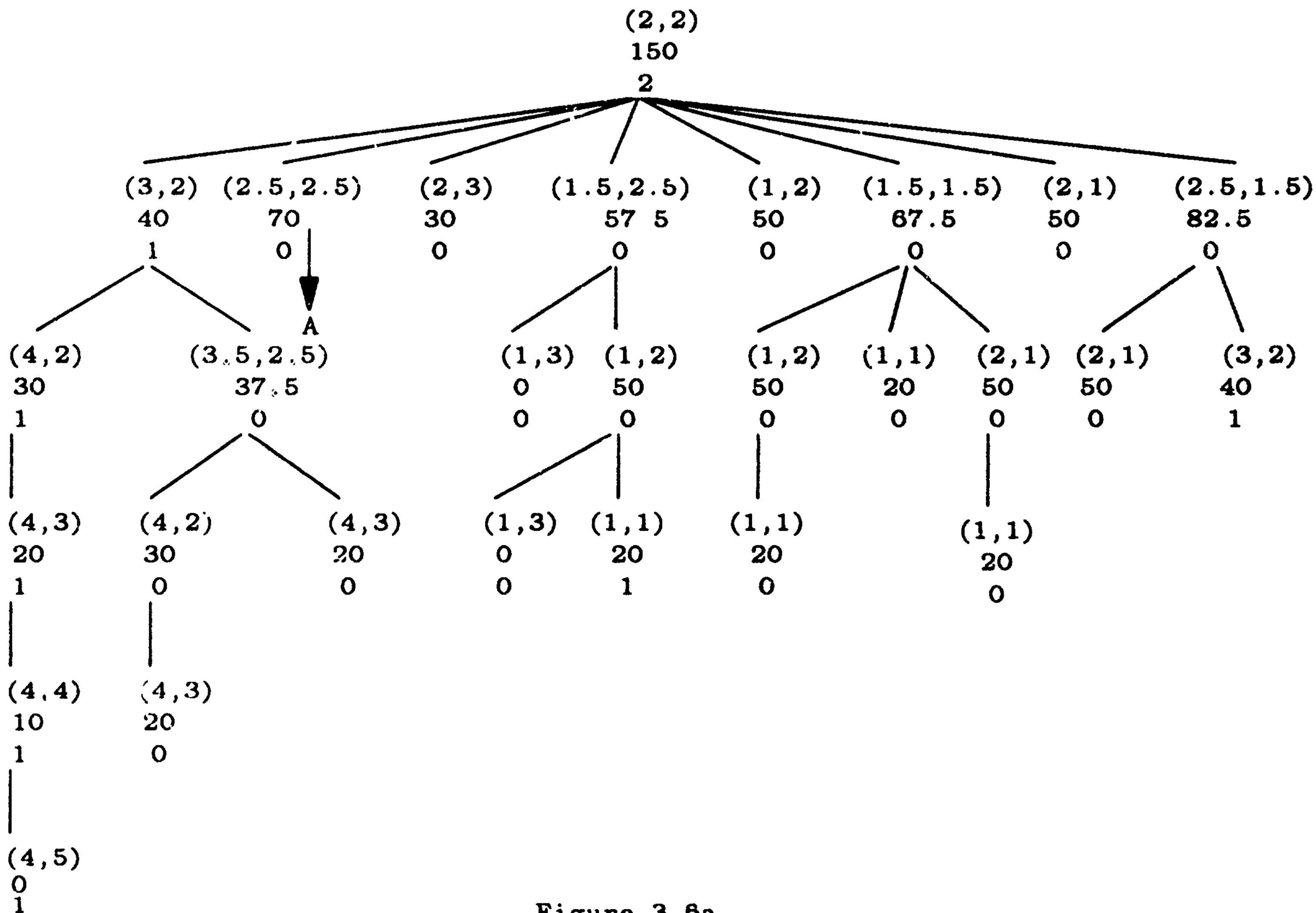


Figure 3.6a
The First Contouring Tree Of The 4x5 Grid

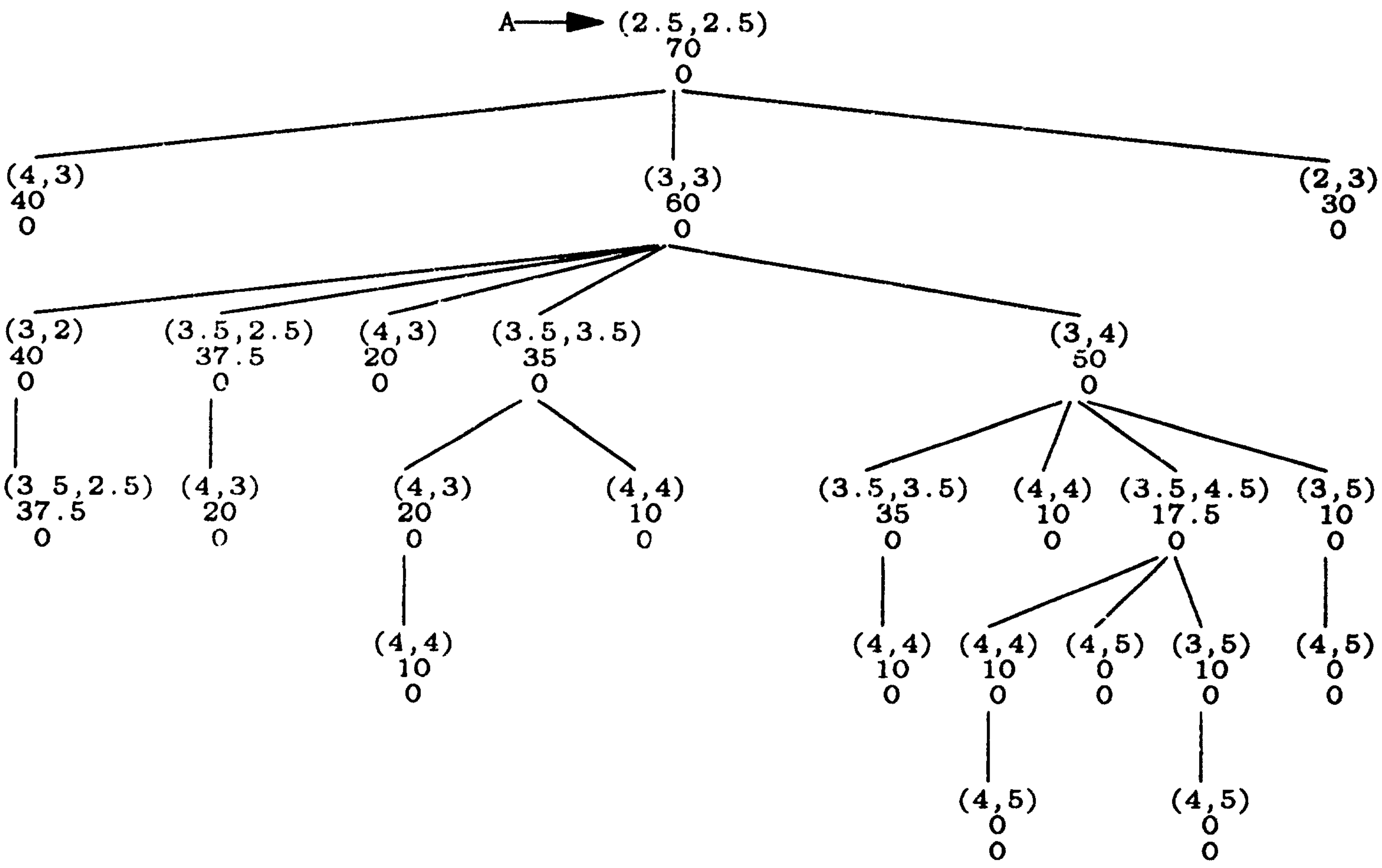


Figure 3.6a (continued)
The First Contouring Tree Of The 4x5 Grid

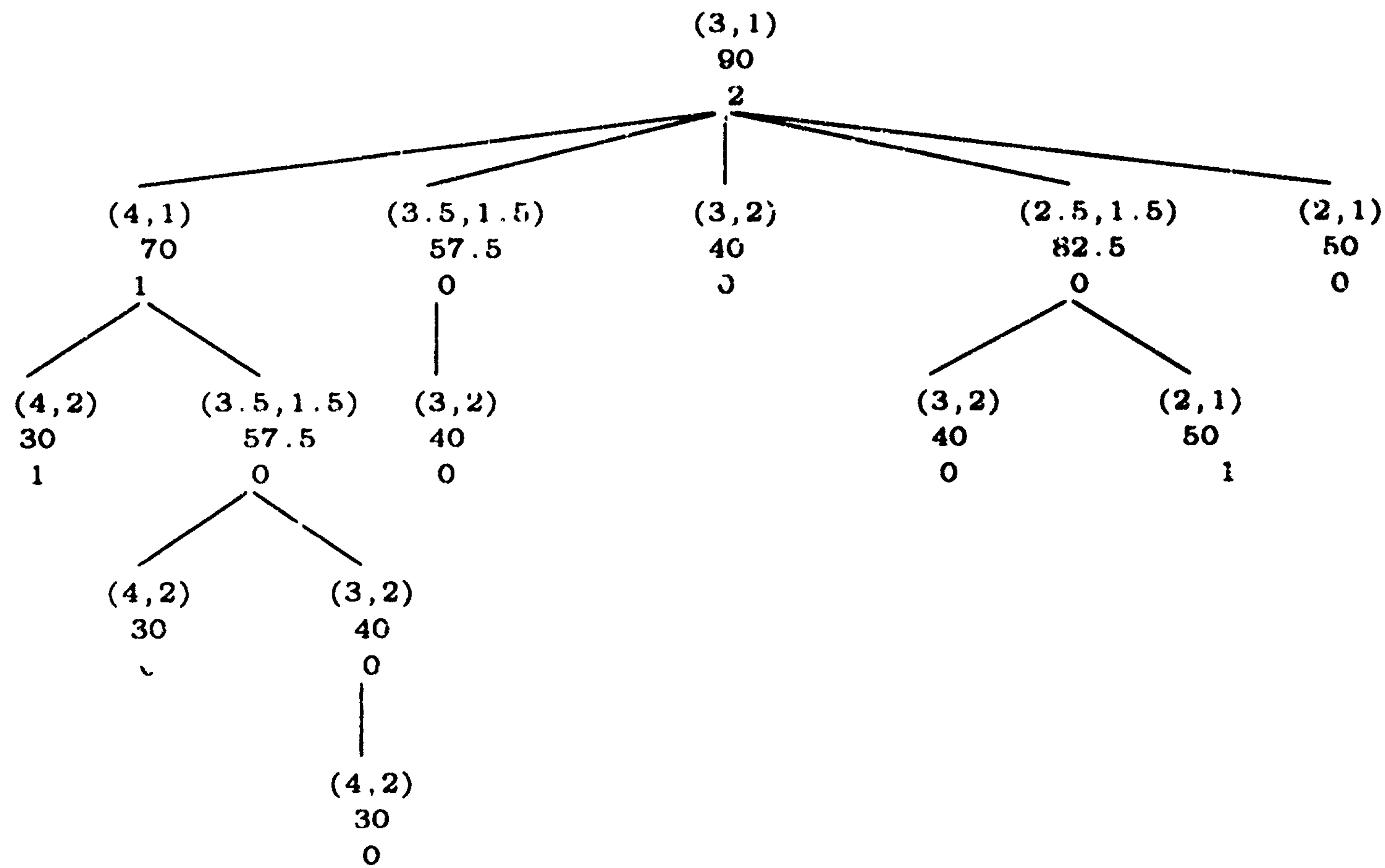


Figure3.6b
The Second Contouring Tree Of The 4x5 Grid

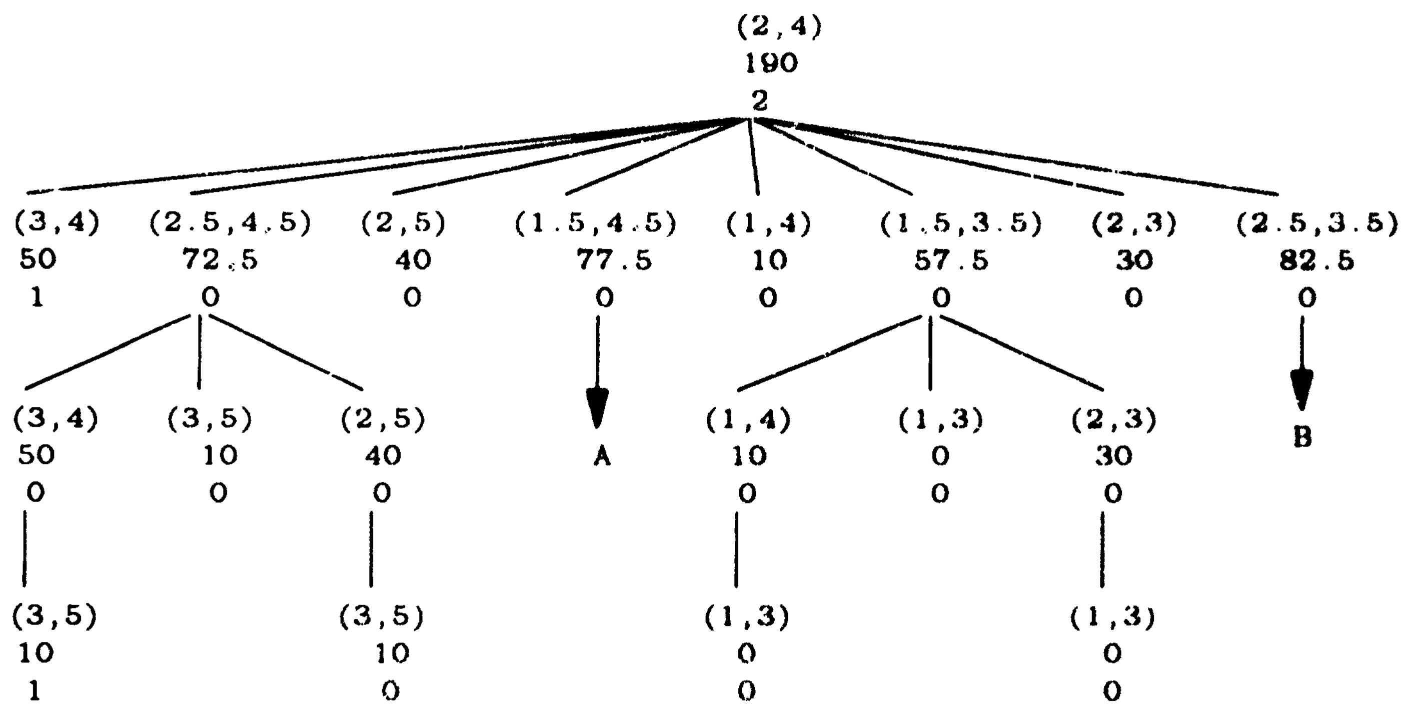


Figure 3.6c
The Third Contouring Tree Of the 4x5 Grid

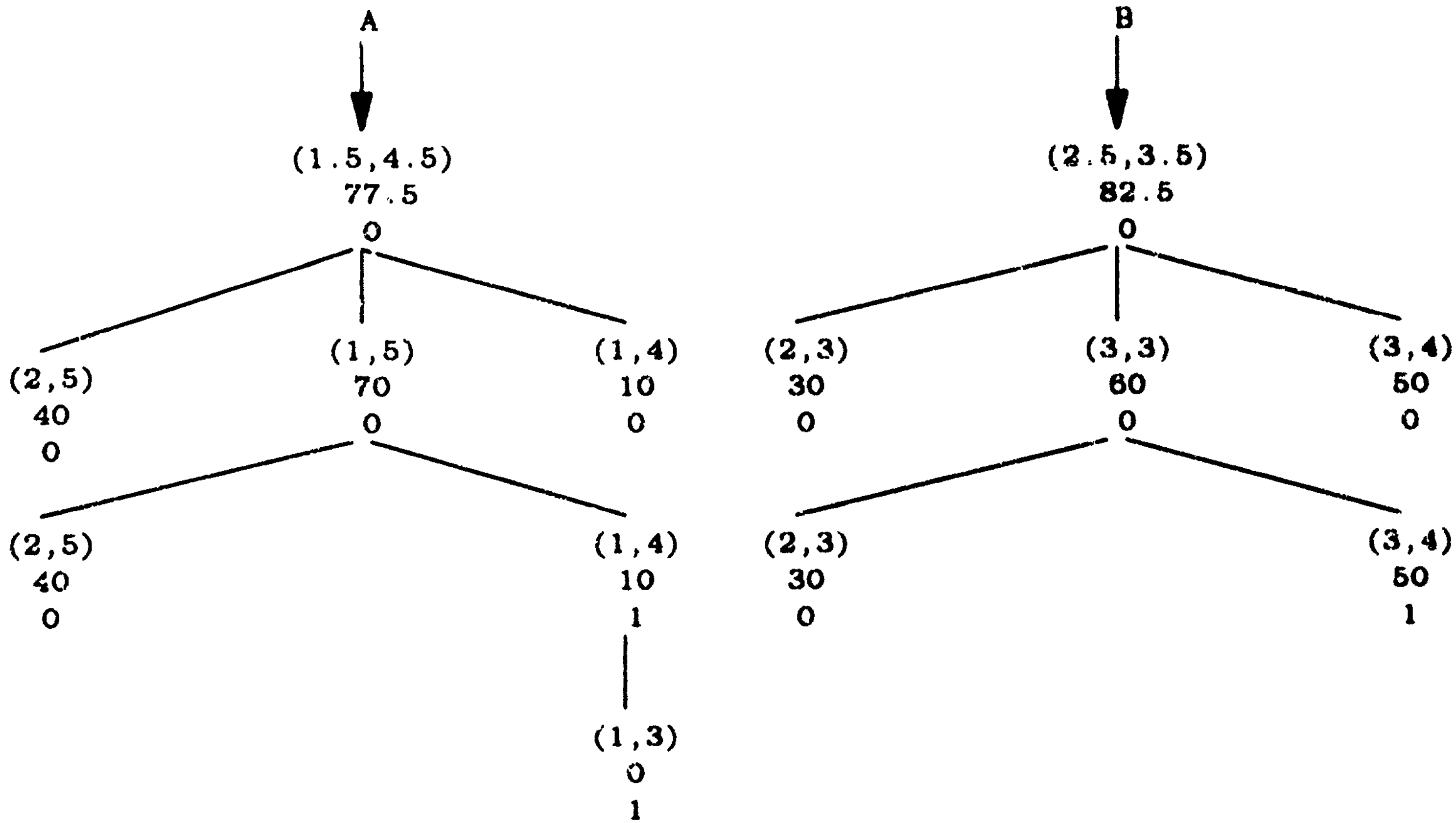


Figure 3.6c (continued)
The Third Contouring Tree Of The 4x5 Grid

come to node (2.5,2.5) and then start checking the edges connected to node (2.5,2.5) in a counterclockwise order. The first edge is edge (2.5,2.5)-(3,2). Using this edge, we come to node (3,2). But we have already visited this node. Now we apply the rule explained above. In this case, we only take care of the edges immediately adjacent to edge (2.5,2.5)-(3,2). We skip the rest of the edges connected to node (3,2). The adjacent edges are edges (3,2)-(2,2) and (3,2)-(3,3). Since these two edges are inward we don't link any nodes to the tree.

C. DRAWING INSTRUCTION PLACEMENT

In the contouring tree for the example grid of *Figure* 3.6, the number under the density value of the node shows the drawing command for that node. We insert drawing commands by way of a pre-order traversal of the contouring tree. The edges are examined in a counterclockwise, and downward ordering from the root. We note that we need to place setpoint drawing commands on the lower valued node of each edge that presents a new lowest value for the tree. *Figure* 3.6 shows the setpoint command "1" under the node coordinate for each new lowest density value in pre-order traversal order. We can also see the setpoint command "1" under the lowest valued node of each edge where split edge problems occur. Some neighboring edges in the contouring tree, i.e. edges sharing an ancestor node, have a "split" between them, i.e., the edges are not immediate counterclockwise neighbors in the original grid. We indicate this split by placing a "1" on the lower valued node of the edge where the discontinuity occurs. For example, in *Figure* 3.4, the edges (1,2)-(1,3) and (1,2)-(1,1) are neighbors in the contouring tree but are not immediate neighbors in the original grid. We place a "1" on the lower valued node of edge (1,2)-(1,1), i.e. node (1,1).

D. DISPLAY GENERATION

Display generation from the contouring trees for the example grid is accomplished by performing a pre-order traversal of those trees, producing a coordinate and drawing instruction whenever the desired contour level is found to be within the range of an edge of a contouring tree. The coordinate for each of these edges is generated by a linear interpolation of the edge's endpoint coordinates according to the decrease in contour level along the edge.

The coordinates and drawing commands generated for the contouring trees of *Figure* 3.6 at level 50 and 100 are shown in *Figure* 3.7. *Figure* 3.8 shows the grid with contours drawn for levels 50 and 100.

First Tree Rooted At Value 150.00			
Level 50			
X	Y	Z	D
2.9091	2.0000	0.0000	1
2.8333	2.1667	0.0000	0
3.0000	2.5000	0.0000	0
3.2222	2.7778	0.0000	0
3.2500	3.0000	0.0000	0
3.2000	3.2000	0.0000	0
3.0000	4.0000	0.0000	0
2.6667	3.0000	0.0000	1
2.2500	2.7500	0.0000	0
2.0000	2.8333	0.0000	0
1.6364	2.6364	0.0000	0
1.4348	2.5652	0.0000	0
1.0000	2.0000	0.0000	0
1.3158	1.3158	0.0000	0
2.0000	1.0000	0.0000	0
2.8824	1.8824	0.0000	1
2.9091	2.0000	0.0000	0

Second Tree Rooted At Value 90.00			
Level 50			
X	Y	Z	D
4.0000	1.5000	0.0000	1
3.6364	1.6364	0.0000	0
3.2857	1.7143	0.0000	0
3.0000	1.8000	0.0000	0
2.8824	1.8824	0.0000	0
2.0000	1.0000	0.0000	1
2.0000	1.0000	0.0000	0

Column D is the drawing command, i.e. 1 = SETPOINT, 0 = DRAWTO

Figure 3.7

Coordinates Generated For The 4X5 Grid

Third Tree Rooted At Value 190.00			
Level 50			
X	Y	Z	D
3.0000	4.0000	0.0000	1
3.0000	4.0000	0.0000	0
2.6800	4.6800	0.0000	0
2.1538	4.8462	0.0000	0
2.0000	4.9333	0.0000	0
1.8667	4.8667	0.0000	0
1.6667	5.0000	0.0000	0
1.0000	4.6667	0.0000	1
1.2963	4.2963	0.0000	0
1.2222	4.0000	0.0000	0
1.4211	3.5789	0.0000	0
1.4348	3.4348	0.0000	0
1.6364	3.3636	0.0000	0
2.0000	3.1250	0.0000	0
2.1905	3.1905	0.0000	0
2.6667	3.0000	0.0000	0
3.0000	4.0000	0.0000	1
3.0000	4.0000	0.0000	0

First Tree Rooted At Value 150.00			
Level 100			
X	Y	Z	D
2.4545	2.0000	0.0000	1
2.3125	2.3125	0.0000	0
2.0000	2.4167	0.0000	0
1.7297	2.2703	0.0000	0
1.5000	2.0000	0.0000	0
1.6970	1.6970	0.0000	0
2.0000	1.5000	0.0000	0
2.3704	1.6296	0.0000	0
2.4545	2.0000	0.0000	0

Second Tree Rooted At Value 90.00			
Level 100			
X	Y	Z	D
No coordinate			

Column D is the drawing command, i.e. 1 = SETPOINT, 0 = DRAWTO

Figure 3.7 (continued)

Coordinates Generated For The 4X5 Grid

Third Tree Rooted At Value 190.00			
Level 100			
X	Y	Z	D
2.6429	4.0000	0.0000	1
2.3830	4.3830	0.0000	0
2.0000	4.6000	0.0000	0
1.6000	4.4000	0.0000	0
1.5000	4.0000	0.0000	0
1.6604	3.6604	0.0000	0
2.0000	3.4375	0.0000	0
2.4186	3.5814	0.0000	0
2.6429	4.0000	0.0000	0

Column D is the drawing command, i.e. 1 = SETPOINT, 0 = DRAWTO

Figure 3.7 (continued)

Coordinates Generated For The 4X5 Grid

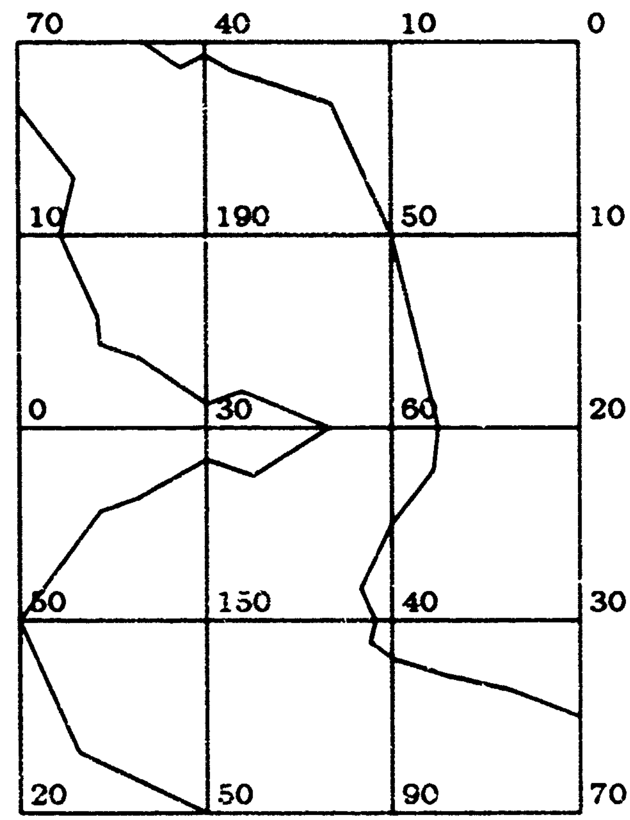


Figure 3.8a
The 4x5 Grid with Contours Drawn for Level 50

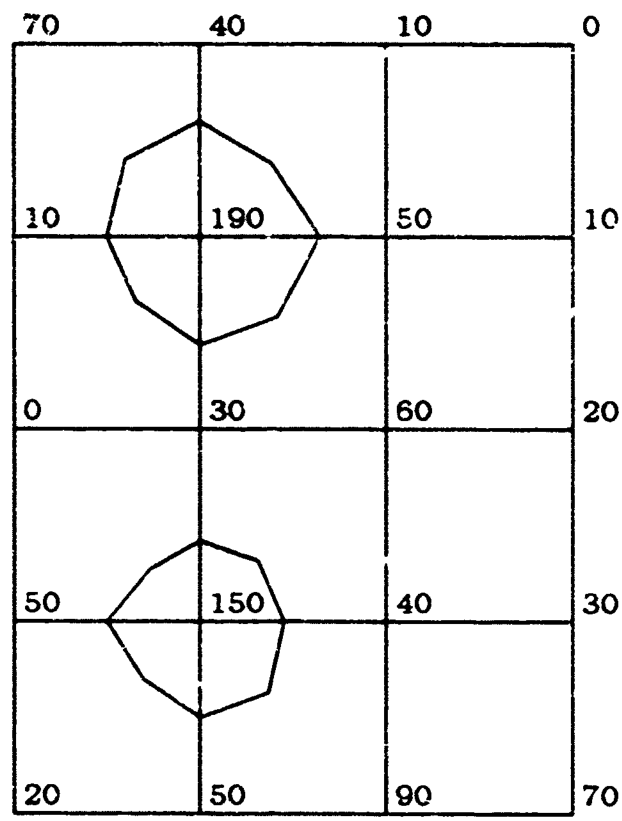


Figure 3 8b
The 4x5 Grid with Contours Drawn for Level 10C

IV. CONCLUSIONS

This study has described a graph theoretic algorithm for contour display generation. A Large Contouring Tree Algorithm for the operations used to generate the contour lines for a regularly subdivided grid was developed.

The inadequacies of currently published algorithms, with respect to contour line generation for a regular grid, have been pointed out in a brief review of the available literature. The new algorithm solves the picture efficiency problems described in [Refs. 1-4], [Refs. 6-8]. A data structure, the Large Contouring Tree has been introduced as the basis of a new algorithm for generating the contour lines for a two-dimensional grid. The presented algorithm is based on the 2x2 Subgrid Algorithm of [Ref. 1]. The 2x2 subgrid algorithm builds a general framework useful for the generation of the coordinates and drawing instructions for any 2x2 subgrid. But there is a picture efficiency problem with this algorithm, i.e. edge duplication and vector ordering problems. The Large Contouring Algorithm solves these problems.

The only problem with the new contouring tree algorithm is when all of the edges in a 2x2 subgrid of the larger grid are equivalued edges. In this case, the algorithm outputs the coordinates of all the edges in the 2x2 subgrid with proper drawing instructions. The contour lines produced look like squares with diagonal crossing lines. This problem can be solved by determining the equivalued 2x2 subgrids in the larger grid before the contouring tree process and then skipping those 2x2 subgrids.

APPENDIX A

IMPLEMENTATION OF LCT NEW ALGORITHM

```
program LargeContouringTree (InpFile,output);

 type

    (* used to store density values of grid *)

    inputtype = array[0..50,0..50] of real;


    InDegreeMatrixtype=array[1..40,1..40] of integer;

   (* Gives the grid coordinates of the node in the in-degree matrix *)

    CoordType = array 1..100] of record
                        X: integer;
                        Y: integer;
                      end;

    (* gives the node number in the in-degree matrix associated with
      the given node in the grid.   *)

    NodeCoor = array 1..10,1..10] of integer;

    PointerType = ^ NodeType;
    (* Data structure of the binary representation of the contouring
      tree *)

    NodeType = record
             Xval,Yval,Dnsty : real;
             Data,Draw : integer;
             Child,Sibling,pred : PointerType;
           end;

    HeadersType = ^ListOfHeader;

    (* A pointer header list holds the root nodes of the trees *)

    ListOfHeader = record
               Tree : PointerType;
               Child : HeadersType;
             end;
  var

    (* Maximum coordinate values of the input grid *)

    Xmax, Ymax :integer;

    ContourLevel,limit:integer;

    (* Maximum node number in the in-degree matrix *)

    limit:integer;
```

```
    (* Xbase and Ybase are used to change coordinate base
      from (1,1) to whatever input data is given. *)

    Xbase,Ybase      :integer;

    NodeDensity      :inputtype;

    InDegreeMatrix   : InDegreeMatrixtype;

    (* Crossreference from the in-degree matrix to the grid *)

    Coord            CoordType;

    (* Crossreference from the grid to the in-degree matrix *)

    FromGridToMatrix : NodeCoor;

    TREE             : PointerType;

    Headers          : HeadersType;

    InpFile          : text;

(*  ==> SECTION 0 : READING AND WRITING INPUT DATA   <==              *)


(**************************************************************************

  Read the maximum coordinate values for the grid and calculating the
  maximum size of the array which holds the grid density values

 **************************************************************************)

procedure Initialize;
   begin
     reset(InpFile);
     read(InpFile,Xmax);read(InpFile.Ymax),
     read(InpFile,Xbase);read(InpFile,Ybase);
     limit:=(Xmax*Ymax-(Xmax-1)*(Ymax-1)).
   end.

(**************************************************************************

  Read the input data and calculating the average density values on
  center points

 **************************************************************************)

procedure ReadData,
  var i,j . integer;
  begin
    for j := 1 to Ymax do
      for i - 1 to Xmax do
         read( InpFile,NodeDensity i,j  );
```

```
    readln(InpFile),

    (* Calculating average density values *)

    for j:= 1 to Ymax-1 do
      for i:= 1 to Xmax-1 do
        NodeDensity[Xmax+i,j]:=(NodeDensity[i+1,j] + NodeDensity[i+1,j-1] +
                         NodeDensity[i,j] + NodeDensity[i,j-1])/4 ;
  end; (* PROC *)

(****************************************************************************

  Write the input data with calculated average density values on center
  points

****************************************************************************)

procedure WriteInpData;
  var i,j : integer;
  begin
    writeln;
    write('*************   DENSITY VALUES OF NODES IN GRID ');
    writeln('**************');
    writeln;
    write('X --> ' 13,'1':3);
    for i := 2 to 2*Xmax - 1 do
      write(i:8);writeln;
    writeln;
    for j := 1 to Ymax do begin
      if j=1
        then write('Y --> ':9,j:3,'-')
      else write(j:12,'-');
      for i := 1 to 2*Xmax - 1 do
           write(NodeDensity[i,j]:8:3);
      writeln; writeln;
    end (* FOR *) ;
    writeln;
    write('Locations Of Average Density Values Start ');
    writeln('After X - ',Xmax:2);
  end ; (* PROC *)

(*        =    SECTION 1   CREATION OF IN-DEGREE MATRIX   -              *)


(****************************************************************************

  Situation 1 is the name of the case for the node in the lower
  lefthand corner of the grid

****************************************************************************)

 procedure Situation1 (i,j : integer;var k,l integer );
   begin

     (* checking edge number 1 *)
```

```
    if NodeDensity[i,j] >= NodeDensity[i+1,j]
      then InDegreeMatrix[i,i+1] := -1
    else InDegreeMatrix[i+1,i]:= -1;

    (* checking edge number 2 *)

    if NodeDensity[i,j] >= NodeDensity[Xmax+i,j]
      then InDegreeMatrix[i,k] := -1
    else InDegreeMatrix[k,i] := -1;

    (* checking edge number 3 *)

    if NodeDensity[i,j] >= NodeDensity[i,j+1]
      then InDegreeMatrix[i,k-1] := -1
    else InDegreeMatrix[k-1,i] := -1;

    (* Increments of the "l" and "k" *0

    l := l+1;
    if ( Xmax <> 2 )
      then k := k+3;
  end (* PROCEDURE *);

(*****************************************************************************

  Situation 2 is the name of the case for all nodes on the perimeter grid
  line with the lowest valued Y coordinate, except for the first and last
  nodes.

*****************************************************************************)

procedure Situation2 (i,j :integer;var k,l:integer );
 begin

   (* Checking edge number 3 *)
   if NodeDensity[i,j] >= NodeDensity[i,j+1] then begin
     if i = 2
       then InDegreeMatrix[i,i+1] := -1
     else InDegreeMatrix[l,k-4] := -1
   end
   else begin
     if i = 2
       then InDegreeMatrix[i+1,i] := -1
     else InDegreeMatrix[k-4,l] := -1
   end;

   (* Checking edge number 4 *)

   if NodeDensity[i,j] >= NodeDensity[Xmax+i-1,j]
     then InDegreeMatrix[l,k-3] := -1
   else InDegreeMatrix[k-3,l] := -1;

   (* Checking edge number 2 *)

   if NodeDensity[i,j] >= NodeDensity[Xmax+i,j]
     then InDegreeMatrix[l,k] := -1
```

```
    else InDegreeMatrix[k,l] := -1;

    (* Checking edge number 1 *)

    if NodeDensity[i,j] >= NodeDensity[i+1,j] then begin
      if i = 2
        then InDegreeMatrix[i.l+4] := -1
      else InDegreeMatrix[l,l+3] := -1
    end
    else begin
      if i = 2
        then InDegreeMatrix[l+4,i] := -1
      else InDegreeMatrix[l+3,l] := -1
    end; (* IF *)

    (* Increments of the "l" and "k" *)

    if i = 2
      then l := l+4
    else l := l+3;
    if i <> ( Xmax-1)
      then k := k+3
end; (* PROC *)

(****************************************************************************

  Situation 3 is the name of the case for the node in the lower righthand
  corner of the grid.

 ****************************************************************************)

procedure Situation3 (i,j :integer;var k.l:integer );
  begin

    (* Checking edge number 1 *)

    if NodeDensity[i,j] >= NodeDensity[i,j+1]
      then InDegreeMatrix[l,l+1] := -1
    else InDegreeMatrix[l+1,l] := -1;

    (* Checking edge number 2 *)

    if NodeDensity i,j  >= NodeDensity[Xmax+i-1,j
      then InDegreeMatrix l.k  = -1
    else InDegreeMatrix k,l] := -1;

    (* Increments of the "l" and "k" *)

    l := 4;
    if Ymax = 2
      then k  = 5
    else k := k-3;
  end . (* PROC *)

(****************************************************************************
```

```
Situation 5 is the name of the case for all nodes on the perimeter grid
line with the lowest valued X coordinate, except for the first and last
nodes.

*****************************************************************************)

procedure Situation5 (i,j :integer;var k,l:integer );
 begin

   (* Checking edge number 2 *)

   if NodeDensity[i,j] >= NodeDensity[Xmax+i,j-1]
     then InDegreeMatrix[l,l-1]:=-1
   else InDegreeMatrix[l+1,l] := -1;

   (* Checking edge number 3 *)

   if NodeDensity[i,j] >= NodeDensity[i+1,j]
    then InDegreeMatrix[l,l-1] := -1
   else InDegreeMatrix[i-1,l] := -1;

   (* Checking edge number 4 *)

   if NodeDensity[i,j] >= NodeDensity[Xmax+i,j]
    then InDegreeMatrix[l,k] := -1
   else InDegreeMatrix[k,l] := -1;

   (* Checking edge number 5 *)

   if NodeDensity[i,j] >= NodeDensity[i,j+1]
    then InDegreeMatrix[l,k-1] := -1
   else InDegreeMatrix[k-1,l] := -1;

   (* Increments of the "l" and "k" *)

   l := l-1;
   if Xmax <> 2
     then k:=k-2;
 end; (* PROC *)

(*****************************************************************************

  Situation 6 is the name of the case for non-perimeter nodes that occur
  at crossing points of the two-dimensional grid.

*****************************************************************************)

procedure Situation6 (i,j :integer;var k,l:integer );
 begin

   (* checking edge number 3 *)

   if NodeDensity[i,j] >= NodeDensity[i,j-1] then begin
    if i = 2
      then InDegreeMatrix[l,k-4] := -1
    else InDegreeMatrix[l,k-3] := -1
```

```
end
else begin
  if i = 2
    then InDegreeMatrix[k-4,l] := -1
  else InDegreeMatrix[k-3,l] := -1
end;

 (* checking edge number 4 *)

if NodeDensity[i,j] >= NodeDensity[Xmax+i-1,j]
  then InDegreeMatrix[l,k-2] := -1
else InDegreeMatrix[k-2,l] := -1;

 (* checking edge number 2 *)

if NodeDensity[i,j] >= NodeDensity[Xmax+i,j]
  then InDegreeMatrix[l,k] := -1
else InDegreeMatrix[k,l] := -1;

 (* checking edge number 6 *)

if NodeDensity[i,j] >= NodeDensity[Xmax+i-1,j-1] then begin
  if i =2
    then InDegreeMatrix[l,l+2] := -1
  else InDegreeMatrix[l,l+1] := -1
end
else begin
  if i = 2
    then InDegreeMatrix[l+2,l] := -1
  else InDegreeMatrix[l+1,l] := -1
end;

 (* checking edge number 8 *)

if NodeDensity[i,j] >= NodeDensity[Xmax+i,j-1] then begin
  if ( i = 2 ) and ( j = 2 )
    then InDegreeMatrix[l,l+5] := -1
  else if ( i = 2 ) or ( j = 2 )
    then InDegreeMatrix[l,l+4] := -1
  else InDegreeMatrix[l,l+3] := -1
end
else begin
  if ( i = 2 ) and ( j = 2 )
    then InDegreeMatrix[l+5,l] := -1
  else if ( i = 2 ) or ( j = 2 )
    then InDegreeMatrix[l+4,l] := -1
  else InDegreeMatrix[l+3,l] := -1
end;

 (* checking edge number 1 *)

if NodeDensity[i,j] >= NodeDensity[i+1,j] then begin
  if ( i = 2 ) and ( j = 2 ) then begin
     InDegreeMatrix[l,l+4] := -1;
     l := l+4
  end
```

```
    else if ( j = 2 ) or ( i = 2 ) then begin
      InDegreeMatrix[l,l+3] := -1;
      l := l+3
    end
    else begin
      InDegreeMatrix[l,l+2] := -1;
      l := l+2
    end
  end
  else begin
    if ( i = 2 ) and ( j = 2 ) then begin
      InDegreeMatrix[l+4,l] := -1;
      l := l+4
    end
    else if ( j = 2 ) or ( i = 2 ) then begin
      InDegreeMatrix[l+3,l] := -1;
      l := l+3
    end
    else begin
      InDegreeMatrix[l+2,l] := -1;
      l := l+2
    end
  end;
  if i <> ( Xmax-1 )
    then k := k+2
end;(* PROC *)

(*****************************************************************************

  Situation 7 is the name of the case for all nodes on the perimeter grid
  line with the highest valued X coordinate, except for the first and
  last nodes

*****************************************************************************)

procedure Situation7 (i,j :integer;var k,l:integer );
 begin

   (* checking edge number 1 *)

   if NodeDensity[i,j] >= NodeDensity[i,j+1]
     then if Xmax=2
           then InDegreeMatrix[l,k-2]:=-1
         else InDegreeMatrix[l,k-1] := -1
   else if Xmax = 2
           then InDegreeMatrix[k-2,l]:=-1
         else InDegreeMatrix[k-1,l] := -1;

   (* checking edge number 2 *)

   if NodeDensity[i,j] >= NodeDensity[Xmax-i+1,j]
     then InDegreeMatrix[l,k] := -1
   else InDegreeMatrix[k,l] := -1;

   (* checking edge number 4 *)
```

```
  if NodeDensity[i,j] >= NodeDensity[Xmax+i-1,j-1]
   then if Xmax=2
          then InDegreeMatrix[l,l+2]:=-1
        else InDegreeMatrix[l,l+1] := -1
  else if Xmax=2
          then InDegreeMatrix[l+2,l]:=-1
        else InDegreeMatrix[l+1,l] := -1 ;

  (* Increments of the "k" and "l" *)

  if Xmax = 2
    then l:=l+4
  else l := l+3;
  if j <> ( Ymax-1)
   then k := k+3
  else k := l+1;
end; (* PROC *)

(****************************************************************************

 Situation 8 is the name of the case for the node in the upper lefthand
 corner of the grid.

****************************************************************************)

procedure Situation8 (i,j :integer;var k,l:integer ),
 begin

  (* Checking edge number 1 *)

  if NodeDensity[i,j] >= NodeDensity[i+1,j]
   then InDegreeMatrix[l,l-1] := -1
  else InDegreeMatrix[l-1,l] := -1;

  (* Checking edge number 2 *)

  if NodeDensity[i,j] >= NodeDensity[Xmax+i,j-1]
   then InDegreeMatrix[l,l+1] := -1
  else InDegreeMatrix[l+1,l] := -1;

  (* Incremenst of the "k" and "l" *)

  l := l-1;
  if ( Ymax = 2 ) and ( Xmax <> 2 )
    then k := k - 3
  else if ( Ymax <>2 ) and ( Xmax <> 2 )
   then k := k-2.
 end : (* PROC *)

(****************************************************************************

 Situation 9 is the name of the case for all nodes on the perimeter grid line
 with the highest valued Y coordinate, except for the first and last nodes

****************************************************************************)
```

```
procedure Situation9 (i,j :integer;var k,l:integer );
 begin

  (* Checking edge number 2 *)

  if NodeDensity[i,j] >= NodeDensity[Xmax+i-1,j-1]
   then begin
    if Ymax = 2
     then InDegreeMatrix[l,k-3] := -1
    else InDegreeMatrix[l,k-2] := -1
  end
  else if Ymax =2
   then InDegreeMatrix[k-3,l]:= -1
  else InDegreeMatrix[k-2,l] := -1;

  (* Checking edge number 4 *)

  if NodeDensity[i,j] >= NodeDensity[Xmax+i,j-1]
   then InDegreeMatrix[l,k] := -1
  else InDegreeMatrix[k,l] := -1;

  (* Checking edge number 5 *)

  if NodeDensity[i,j] >= NodeDensity[i+1,j]
   then InDegreeMatrix[l,k-1] := -1
  else InDegreeMatrix[k-1,l] := -1;

  (* Increments of the "l" and "k" *)

  if i < ( Xmax-1 ) then begin
   if Ymax = 2 then begin
    k:=k+3;
    l:=k-4;
   end
   else begin
    k := k+2;
    l := k-3;
   end
  end
  else l := k-1;
 end; (* PROC *)

(*****************************************************************************

  Situation 10 is the name of the case for the node in the upper lefthand
  corner of the grid

*****************************************************************************)

procedure Situation10 (i,j :integer;var k,l:integer );
 begin

  (* Checking edge number 2 *)

  if NodeDensity[i,j] >= NodeDensity[Xmax+i-1,j-1]
   then inDegreeMatrix[l,k] := -1
```

```
  else InDegreeMatrix[k,l] := -1
 end ;

(****************************************************************************

  Returns one of the ten possible situations with respect to the given
  coordinate X,Y.

****************************************************************************)

function Find(X,Y :integer ) : integer;
 begin
  if ( X = 1 ) and ( Y = 1 )
    then Find := 1
  else if (X < Xmax ) and ( Y = 1 )
    then Find := 2
  else if ( X = Xmax ) and ( Y = 1 )
    then Find := 3
  else if X > Xmax
    then Find := 4
  else if ( X = 1 ) and not( Y = Ymax )
    then Find := 5
  else if not( ( X = Xmax ) or ( Y = Ymax ))
    then Find := 6
  else if ( X = Xmax ) and not( Y = Ymax )
    then Find := 7
  else if ( X = 1 ) and ( Y = Ymax )
    then Find := 8
  else if not( X = Xmax ) and ( Y = Ymax )
    then Find := 9
  else Find := 10;
 end ;(* FUNC *)

(****************************************************************************

 Calculates the diagonal values of the In-Degree Matrix

****************************************************************************)

procedure CompleteInDegreeMatrix;
 var i,j,Count :integer;
 begin
   for j := 1 to limit do begin
     Count:= 0;
     for i := 1 to limit do

       if ( InDegreeMatrix[i,j] = -1 )
          then Count:= Count+ 1;
            (* Shows how many -1 values are there in each
               column in In-Degree Matrix              *)

     InDegreeMatrix[j,j] := Count;
     FromGridToMatrix[Coord[j].X,Coord[j].Y]:=j;
   end (* FOR *);
  end; (* PROC *)
```

```
(*************************************************************************

  This is the procedure for creating the In-Degree Matrix by calling
  the procedures above.

**************************************************************************)

procedure CreateIndMatrix;
  var i,j,k,k1,l . integer;
  begin

    k:=5;

    l:=1;

    (* Used to createn the crossreference *)

    k1:=1;

    for j := 1 to Ymax do begin
      for i := 1 to Xmax do begin
        Coord[l].X := i;
        Coord[l].Y := j;
        if k > k1 then begin
          Coord[k].X := Xmax+i;
          Coord[k].Y := j;
          k1 := k;
        end;
        case Find(i,j) of
          1:Situation1(i,j,k,l);
          2:Situation2(i,j,k,l);
          3:Situation3(i,j,k,l);
          4:;
          5:Situation5(i,j,k,l);
          6:Situation6(i,j,k,l);
          7:Situation7(i,j,k,l);
          8:Situation8(i,j,k,l);
          9:Situation9(i,j,k,l),
          10:Situation10(i,j,k,l);
        end; (* CASE *)
      end (* FOR *)
    end; (* FOR *)
    CompleteInDegreeMatrix;
  end; (* PROC *)

(*************************************************************************

  Output the In-Deg-ee Matrix.

**************************************************************************)

procedure WriteIndMatrix;
  var i,j : integer;
  begin
    writeln;
    writeln;writeln(' INDEGREE-MATRIX ':35);writeln;writeln;
```

```
  for i := 1 to limit do begin
    write(i:2,')':1);
    for j := 1 to limit do
      write(InDegreeMatrix[i,j]:3);
    writeln;
  end;(* FOR *)
end; (* PROC *)

(*  ===> SECTION 2 :  BUILDING CONTOURING TREE  <===          *)


(*****************************************************************************

  Create the Large Contouring Tree for each root node in the in-degree
  matrix.

*****************************************************************************)

procedure CreateTree;
  var
    X1,Y1,i,T: integer;

    (* For keeping track of the already visited node *)

    CheckNode:array[1..40] of 0..1;

   (* Used to show which field of node will be used to link
      for the next available growth node in the contouring tree *)

    state:1..2;

    (* Used to catch the root node of each contouring tree *)

    first,second : boolean ;

     (* Takes true when all edges on the edge list  are checked,
        otherwise false  *)

    LastEdge : boolean ;


    Head:HeadersType;

(*****************************************************************************

  Put the root node of the contouring tree on the header pointer list

*****************************************************************************)

procedure LinkOneHeader(Root:PointerType);
 begin
  if T > 1 then begin
    new(Head^.Child);
    Head^.Child^.Tree := Root;
    if T=2 then Headers:=Head;
```

```
      Head:=Head^.Child;
    end
    else begin
      new(Head);
      Head^.Tree := Root;
      Headers:=Head;
    end;
  end ;(* PROC *)

(****************************************************************************

  Create the new growth node pointer, put data on it, link this new
  growth node to the contouring tree, depending upon the value of state.

****************************************************************************)

procedure LinkOneNode(X,Y : integer);
  var Temp : PointerType;
  begin
    new(Temp);
    Temp^.Data := FromGridToMatrix(X,Y);
    CheckNode[FromGridToMatrix(X,Y)]:=1;

    if X > Xmax then begin
      Temp^.Xval:= (X-Xmax)-0.5+Xbase;
      Temp^.Yval := Y+0.5+Ybase;
    end
    else begin
      Temp^.Xval:= X+Xbase;
      Temp^.Yval := Y+Ybase;
    end;

    Temp^.Dnsty:=NodeDensity[X,Y];

    case state of
      1 : begin

          if second then begin
            Temp^.pred := TREE;
            TREE^.Child := Temp;
            LinkOneHeader(TREE);
            TREE := TREE^.Child;
            second := false;
          end
          else if first then begin
            TREE := Temp;
            TREE^.pred := nil;
            second := true;
          end
          else begin
            Temp^.pred := TREE;
            TREE^.Child := Temp;
            TREE := TREE^.Child;
          end;
        end ; (* CASE 1 *)
     2 : begin
```

```
        Temp^.pred := TREE;
        TREE^.Sibling := Temp;
        TREE := TREE^.Sibling;
        state := 1;
      end ;
  end; (* CASE *)
  TREE^.Sibling := nil;
  TREE^.Child := nil;
 end ; (* PROC *)

(*****************************************************************************

  Search the immediate father node of the given a node,
  switche the pointer to the father node.

 *****************************************************************************)

procedure SearchFather(var TREE:PointerType);
  begin
    repeat
      if ( TREE^.Sibling <> nil) then begin
        if TREE^.Sibling^.Data = -1 then begin
          dispose(TREE^.Sibling);
          TREE^.Sibling := nil;
          TREE := TREE^.pred;
        end
        else TREE := TREE^.pred;
      end ;
    until TREE^.Sibling = nil;
  end;(* PROC. *)

(*****************************************************************************

  This is the procedure for linking the new growth node to the contouring
  tree. The procedure does this by calling the procedures explained above.
  It also maintains where the next growth node will be linked.

 *****************************************************************************)

procedure ConstructTree(X,Y:integer;ExistWay:boolean);
  begin
   if first
    then LinkOneNode(X,Y)
   else if ExistWay and LastEdge then begin


     case state of
       1: begin
          LinkOneNode(X,Y);
          state := 2;
        end;
       2: begin
          LinkOneNode(X,Y);
          new(TREE^.Sibling);
          TREE^.Sibling^.Data := -1;
```

```
        TREE^.Sibling^.pred := TREE;
        state := 1;
      end;
    end; (* CASE *)
   LastEdge := false;
  end
  else if ExistWay
    then LinkOneNode(X,Y)
  else if LastEdge then begin

   case state of
    1: begin
        SearchFather(TREE);
        state := 2;
      end;
    2: begin
        TREE := TREE^.pred;
        SearchFather(TREE);
      end;
   end; (* CASE *)
   LastEdge := false;
  end;
 end;(* PROC. *)

(*****************************************************************************

  Link the edges immediately adjacent to the edge used to come to
  the node if those edges are outward.

 *****************************************************************************)

procedure SharedEdge(Position,X,Y,Nm,Pointer:integer;var ExistWay:boolean);
var X1,Y1,X2,Y2:integer;
begin
 case Position of
   1 : case Pointer of
       1,3 : begin
              X1:=Xmax+1;
              Y1:=1;
            end;
        2 : begin
              X1:=X;
              Y1:=Y+1;
              X2:=X+1;
              Y2:=Y;
            end;
      end; (* CASE 1 *)
   2 : case Pointer of
        1 : begin
              X1:=Xmax+X;
              Y1:=Y;
            end;
        2 : begin
              X1:=X,
              Y1:=Y+1;
              X2:=X+1;
```

```
              Y2:=Y;
            end;
          3 : begin
              X1:=Xmax+X-1;
              Y1:=Y;
              X2:=Xmax+X;
              Y2:=Y;
            end;
          4 : begin
              X1:=X-1;
              Y1:=Y;
              X2:=X;
              Y2:=Y+1;
            end;
          5 : begin
              X1:=Xmax+X-1;
              Y1:=Y;
            end;
      end; (* CASE 2 *)
   3 : case Pointer of
          1 : begin
              X1:=Xmax+X-1;
              Y1:=Y;
            end;
          2 : begin
              X1:=X-1;
              Y1:=Y;
              X2:=X;
              Y2:=Y+1;
            end;
          3 : begin
              X1:=Xmax+X-1;
              Y1:=Y;
            end;
      end; (* CASE 3 *)
   4 : case Pointer of
          1 : begin
              X1:=X-Xmax+1;
              Y1:=Y;
              X2:=X-Xmax;
              Y2:=Y+1;
            end;
          2 : begin
              X1:=X-Xmax+1;
              Y1:=Y+1;
              X2:=X-Xmax;
              Y2:=Y;
            end;
          3   begin
              X1:=X-Xmax.
              Y1:=Y+1;
              X2:=X-Xmax+1:
              Y2:=Y;
            end;
          4 . begin
              X1:=X-Xmax:
```

```
          Y1:=Y;
          X2:=X-Xmax+1;
          Y2:=Y+1;
        end;
    end; (* CASE 4 *)
  5 : case Pointer of
      1 : begin
          X1:=X+Xmax;
          Y1:=Y-1;
        end;
      2 : begin
          X1:=X+1;
          Y1:=Y;
          X2:=X;
          Y2:=Y-1;
        end;
      3 : begin
          X1:=X+Xmax;
          Y1:=Y;
          X2:=X+Xmax;
          Y2:=Y-1;
        end;
      4 : begin
          X1:=X;
          Y1:=Y+1;
          X2:=X+1;
          Y2:=Y-1;
        end;
      5 : begin
          X1:=Xmax+X;
          Y1:=Y;
        end;
    end; (* CASE 5 *)
  6 : case Pointer of
      1 : begin
          X1:=X+Xmax;
          Y1:=Y,
          X2:=X+Xmax;
          Y2:=Y-1;
        end;
      2 : begin
          X1:=X;
          Y1:=Y+1;
          X2:=X+1;
          Y2:=Y;
        end;
      3 : begin
          X1:=Xmax+X-1;
          Y1:=Y;
          X2:=Xmax+X;
          Y2:=Y;
        end;
      4 : begin
          X1:=X-1;
          Y1:=Y;
          X2:=X;
```

```
            Y2:=Y+1;
          end;
        5 : begin
            X1:=Xmax+X-1;
            Y1:=Y-1;
            X2:=Xmax+X-1,
            Y2:=Y;
          end;
        6 : begin
            X1:=X;
            Y1:=Y-1;
            X2:=X-1;
            Y2:=Y;
          end;
        7 : begin
            X1:=Xmax+X;
            Y1:=Y-1;
            X2:=Xmax+X-1;
            Y2:=Y-1;
          end;
        8 : begin
            X1:=X+1;
            Y1:=Y;
            X2:=X;
            Y2:=Y-1;
          end;
    end; (* CASE 6 *)
  7 : case Pointer of
        1 : begin
            X1:=X+Xmax-1;
            Y1:=Y;
          end;
        2 : begin
            X1:=X-1;
            Y1:=Y;
            X2:=X;
            Y2:=Y-1;
          end;
        3 : begin
            X1:=Xmax+X-1,
            Y1:=Y-1;
            X2:=Xmax+X-1,
            Y2:=Y;
          end;
        4 : begin
            X1:=X;
            Y1:=Y-1.
            X2:=X-1.
            Y2:=Y,
          end;
        5 : begin
            X1:=Xmax+X-1.
            Y1:=Y-1.
          end
    end; (* CASE 7 *)
  8 : case Pointer of
```

```
      1 . begin
            X1:=X+Xmax;
            Y1:=Y-1;
          end;
      2 : begin
            X1:=X+1;
            Y1:=Y;
            X2:=X;
            Y2:=Y-1;
          end;
      3 : begin
            X1:=X+Xmax;
            Y1:=Y-1;
          end;
    end; (* CASE 8 *)
 9 : case Pointer of
      1 : begin
            X1:=X+Xmax-1;
            Y1:=Y-1;
          end;
      2 : begin
            X1:=X;
            Y1:=Y-1;
            X2:=X-1;
            Y2:=Y;
          end;
      3 : begin
            X1:=Xmax+X;
            Y1:=Y-1;
            X2:=Xmax+X-1;
            Y2:=Y-1:
          end;
      4 : begin
            X1:=X+1;
            Y1:=Y;
            X2:=X;
            Y2:=Y-1;
          end:
      5 : begin
            X1:=X+Xmax;
            Y1:=Y-1;
          end;
    end; (* CASE 9 *)
10 : case Pointer of
      1,3 : begin
            X1:=X+Xmax-1;
            Y1:=Y-1;
          end;
      2   begin
            X1:=X;
            Y1:=Y-1;
            X2:=X-1;
            Y2:=Y;
          end;
    end, (* CASE 10 *)
end; (* CASE POSITION *)
```

```
  if InDegreeMatrix[Nm,FromGridToMatrix[X1,Y1]] = -1 then begin
    LastEdge:=true;
    ConstructTree(X1,Y1,ExistWay);
  end;
  if X2 <> 0 then begin
    if InDegreeMatrix[Nm,FromGridToMatrix[X2,Y2]]=-1 then begin
      LastEdge:=true;
      ConstructTree(X2,Y2,ExistWay);
      ExistWay:=false;
      LastEdge:=true;
      ConstructTree(X2,Y2,ExistWay);
    end
    else begin
      ExistWay:=false;
      LastEdge:=true;
      ConstructTree(X2,Y2,ExistWay);
    end;
  end
  else begin
    ExistWay:=false;
    LastEdge:=true;
    ConstructTree(X2,Y2,ExistWay);
  end;
end; (* PROC *)

(*****************************************************************************

  SearchPath searchs all possible paths from the root node to the other
  nodes in the directed graph. This procedure also links the grwoth nodes
  on the paths to the contouring tree by calling the procedure
  "ConstructTree"

*****************************************************************************)

procedure SearchPath(Situation,X,Y,Nm,Pt:integer);
  var
    (* Used to reorder the edges on the list in a counterclockwise *)

    i   :integer;

    (* Takes value "0" if the node is the root node of the tree,
       otherwise "1"                              *)

    I   0..1;

    Num  integer;

    (* Edge pointer points the edge number of the edge
      to be checked      *)

    P   :integer;

    (* Takes true if the edge is outward,  otherwise false *)

    ExistWay : boolean;
```

```
begin

  if InDegreeMatrix[Nm,FromGridToMatrix[X.Y]] = -1 then begin

    Nm:=FromGridToMatrix[X,Y];
    ExistWay := true
  end
  else ExistWay:=false;
  if ( CheckNode[FromGridToMatrix[X,Y]]=1) and (ExistWay)  then begin
    ConstructTree(X,Y.ExistWay);
    SharedEdge(Situation,X,Y,Nm,Pt,ExistWay);
  end
  else begin
    ConstructTree(X,Y,ExistWay);
  if ExistWay or first then begin
    if first then begin

      (* Initial condition *)

      P := 0;
      I := 1;
      first:= false;
    end
    else begin
      I := 0;

      (* Starting value of the edge number pointer *)

      P := Pt

    end:
    Num:=Nm;

    (* Possible situations procedure can go at calling time  *)

    case Situation   of
      1: begin

          for i := 1 to I+2 do begin

            (* putting the edges on the list in a counterclockwise
              order *)

            if P= 3
              then P := 1
            else P  = P+ 1:

          (* If the last edge is checked, then "LastEdge" returns true *)

            if i = I + 2 then LastEdge  = true;
            case P of

              (* The first possible path using edge number 1 in
                Situation 1 *)

              1  if ( X+1) = Xmax (* First possible route from "1" *)
```

```
          then SearchPath(3,X+1,Y,Num,3)
         else SearchPath(2,X+1,Y,Num,5);

      (* The second possible path using edge number 1 in
         Situation 1 *)

      2 : SearchPath(4,Xmax+X,Y,Num,1);(* Second possible route
                                          from situation "1" *)

      (* The third possible path using edge number 1 in

      3 : if (Y+1) = Ymax (* Third possible route from situ."1*)
           then SearchPath(8,X,Y+1,Num,1)
          else  SearchPath(5,X,Y+1,Num,1);
    end; (* CASE *)
  end; (* FOR *)
 end;
2: begin
   for i := 1 to I+4 do begin
     if P= 5
       then P := 1
     else P := P+ 1;
     if i = I + 4 then LastEdge := true;
     case P of
       1 : if ( X+1) = Xmax
            then SearchPath(3.X+1.Y,Num,3)
           else SearchPath(2,X+1,Y,Num,5) ;
       2  SearchPath(4,Xmax+X,Y,Num,1);
       3 : if (Y+1) =Ymax
            then SearchPath(9,X,Y+1,Num,3)
           else SearchPath(6,X,Y+1.Num.7);
       4 · SearchPath(4,Xmax+X 1,Y,Num.2);
       5 . if ( X-1) = 1
            then SearchPath(1,X-1,Y,Num,1)
           else SearchPath(2,X-1,Y,Num,1)
     end; (* CASE *)
   end; (* FOR *)
 end;
3: begin
   for i := 1 to I+2 do begin
     if P= 3
      then P := 1
     else P := P+ 1;
     if i = I + 2 then LastEdge := true;
     case P of
       1  if ( Y+1) = Ymax
            then SearchPath(10,X,Y+1,Num,3)
           else SearchPath(7,X,Y+1.Num,5) ;
       2  SearchPath(4,Xmax+X-1,Y,Num.2);
       3  if ( X-1) = 1
            then SearchPath(1,X-1,Y,Num,1)
           else SearchPath(2,X-1,Y,Num,1)
     end; (* CASE *)
   end; (* FOR *)
 end;
4: begin
```

```
    for i := 1 to I+3 do begin
      if P= 4
        then P := 1
      else P := P+ 1;
      if i = I + 3 then LastEdge := true;
      case P of
        1 : case Find(X-Xmax,Y) of
              1 : SearchPath(1,X-Xmax,Y,Num,2);
              2 : SearchPath(2,X-Xmax,Y,Num,2);
              5 : SearchPath(5,X-Xmax,Y,Num,4);
              6 : SearchPath(6,X-Xmax,Y,Num,2);
            end ; (* CASE *)
        2 : case Find(X-Xmax+1,Y) of
              2 : SearchPath(2,X-Xmax+1,Y,Num,4);
              3 : SearchPath(3,X-Xmax+1,Y,Num,2);
              6 : SearchPath(6,X-Xmax+1,Y,Num,4);
              7 : SearchPath(7,X-Xmax+1,Y,Num,2);
            end ; (* CASE *)
        3 : case Find(X-Xmax+1,Y+1) of
              6 : SearchPath(6,X-Xmax+1,Y+1,Num,6);
              7 : SearchPath(7,X-Xmax+1,Y+1,Num,4);
              9 : SearchPath(9,X-Xmax+1,Y+1,Num,2);
             10 : SearchPath(10,X-Xmax+1,Y+1,Num,2);
            end ; (* CASE *)
        4 : case Find(X-Xmax,Y+1) of
              5 : SearchPath(5,X-Xmax,Y+1,Num,2);
              6 : SearchPath(6,X-Xmax,Y+1,Num,8);
              8 : SearchPath(8,X-Xmax,Y+1,Num,2);
              9 : SearchPath(9,X-Xmax,Y+1,Num,4);
            end ; (* CASE *)
      end; (* CASE *)
    end; (* FOR *)
  end;
5: begin
    for i := 1 to I+4 do begin
      if P= 5
        then P := 1
      else P := P+ 1;
      if i = I + 4 then LastEdge := true;
      case P of
        1 : if ( Y - 1 ) = 1
              then SearchPath(1,X,Y-1,Num,3)
            else SearchPath(5,X,Y-1,Num,5) ;
        2 : SearchPath(4,Xmax+X,Y-1,Num,4);
        3 : if ( X + 1 ) = Xmax
              then SearchPath(7,X+1,Y,Num,3)
            else SearchPath(6,X+1,Y,Num,5);
        4 : SearchPath(4,Xmax+X,Y,Num,1);
        5 : if ( Y + 1 ) = Ymax
              then SearchPath(8,X,Y+1,Num,1)
            else SearchPath(5,X,Y+1,Num,1)
      end; (* CASE *)
    end; (* FOR *)
  end;
6: begin
    for i := 1 to I+7 do begin
```

```
        if P= 8
          then P := 1
        else P := P+ 1;
        if i = I + 7 then LastEdge := true;
        case P of
          1 : if ( X + 1 ) =Xmax
                then SearchPath(7,X+1,Y,Num,3)
              else SearchPath(6,X+1,Y,Num,5) ;
          2 : SearchPath(4,Xmax+X,Y,Num,1);
          3 : if ( Y + 1 ) =Ymax
                then SearchPath(9,X,Y+1,Num,3)
              else SearchPath(6,X,Y+1,Num,7) ;
          4 : SearchPath(4,Xmax+X-1,Y,Num,2);
          5 : if ( X - 1 ) =1
                then SearchPath(5,X-1,Y,Num,3)
              else SearchPath(6,X-1,Y,Num,1) ;
          6 : SearchPath(4,Xmax+X-1,Y-1,Num,3);
          7 : if ( Y - 1 ) =1
                then SearchPath(2,X,Y-1,Num,3)
              else SearchPath(6,X,Y-1,Num,3) ;
          8 : SearchPath(4,Xmax+X,Y-1,Num,4);
        end; (* CASE *)
      end; (* FOR *)
    end;
  7: begin
      for i := 1 to I+4 do begin
        if P= 5
          then P := 1
        else P := P+ 1;
        if i = I + 4 then LastEdge := true;
        case P of
          1 : if ( Y - 1 ) = Ymax
                then SearchPath(10,X,Y+1,Num,3)
              else SearchPath(7,X,Y+1,Num,5);
          2 : SearchPath(4,Xmax+X-1,Y,Num,2);
          3 : if ( X - 1 ) = 1
                then SearchPath(5,X-1,Y,Num,3)
              else SearchPath(6,X-1,Y,Num,1);
          4 : SearchPath(4,Xmax+X-1,Y-1,Num,3);
          5 : if ( Y - 1 ) = 1
                then SearchPath(3,X,Y-1,Num,1)
              else SearchPath(7,X,Y-1,Num,1);
        end; (* CASE *)
      end; (* FOR *)
    end;
  8: begin
      for i := 1 to I+2 do begin
        if P= 3
          then P := 1
        else P := P+ 1;
        if i = I + 2 then LastEdge := true;
        case P of
          1 : if ( Y - 1 ) = 1
                then SearchPath(1,X,Y-1,Num,3)
              else SearchPath(5,X,Y-1,Num,5) ;
          2 : SearchPath(4,Xmax+X,Y-1,Num,4);
```

```
              3 : if ( X - 1 ) = Xmax
                   then SearchPath(10,X+1,Y,Num,1)
                 else SearchPath(9,X+1,Y,Num,1) ;
            end; (* CASE *)
          end; (* FOR *)
        end;
      9: begin
          for i := 1 to I+4 do begin
            if P= 5
              then P := 1
            else P := P+ 1;
            if i = I + 4 then LastEdge := true;
            case P of
              1 : if ( X - 1 ) = 1
                   then SearchPath(8,X-1,Y,Num,3)
                 else SearchPath(9,X-1,Y,Num,5);
              2 . SearchPath(4,Xmax+X-1,Y-1,Num,3);
              3 : if ( Y - 1 ) = 1
                   then SearchPath(2,X,Y-1,Num,3)
                 else SearchPath(6,X,Y-1,Num,3) ;
              4 : SearchPath(4,Xmax+X,Y-1,Num,4);
              5 : if ( X + 1 ) = Xmax
                   then SearchPath(10,X+1,Y,Num,1)
                 else SearchPath(9,X+1,Y,Num,1) ;
            end; (* CASE *)
          end; (* FOR *)
        end;
      10: begin
          for i := 1 to I+2 do begin
            if P= 3
              then P := 1
            else P := P+ 1;
            if i = I + 2 then LastEdge := true;
            case P of
              1 : if ( X - 1 ) = 1
                   then SearchPath(8,X-1,Y,Num,3)
                 else SearchPath(9,X-1,Y,Num,5);
              2 : SearchPath(4,Xmax+X-1,Y-1,Num,3);
              3 : if ( Y - 1 ) = 1
                   then SearchPath(3,X,Y-1,Num,1)
                 else SearchPath(7,X,Y-1,Num,1) ;
            end; (* CASE *)
          end; (* FOR *)
        end;
    end;
  end ; (* IF *)
 end; (* ELSE IF *)
end ; (* PROC *)

(*************************************************************************

  Procedure "CreateTree" -continued-

*************************************************************************)

begin
```

```
  T:=0;
  for i:=1 to limit
     do CheckNode[i]:=0;
  for i:=1 to limit do begin
    if InDegreeMatrix[i,i] = 0 then begin

       (* Root is recognized in In_Degree Matrix *)

       state := 1;
       T:=T+1;
       first := true;
       second := false;

       (* Coordinate of root *)

       X1 := Coord[i].X;
       Y1 := Coord[i].Y;
       SearchPath(Find(X1,Y1),X1,Y1,i,0);
    end; (* IF *)
  end; (* FOR *)
  Head^.Child:=nil;
end; (* PROC *)


(* ====> SECTION 3 : INSERTION DRAWING COMMAND  ON NODES  <====        *)


(*****************************************************************************

  In this function, first the coordinates of the edge which exists
  in the contouring tree are found. Second the coordinates of the edges
  which should exist in the contouring tree for continuity is found.
  If those coordinates are the same, then there is no split edge problem.
  Otherwise there is.

*****************************************************************************)

function Adjacent(Tree:PointerType):boolean;
 var Xr,Xt,Xs,Yr,Yt,Ys : integer;
     Root : PointerType;
 begin
  Root := Tree;

  (* Finding the root pointer of the given node *)

  while ( Root = Root^.pred^.Sibling ) do
     Root := Root^.pred;
  Root := Root^.pred;

  (* Coordinates of the root node of the given subtree *)

  Xr := Coord[Root^.Data].X;
  Yr := Coord[Root^.Data].Y;

  (* Coordinates of the given node as a parameter *)
```

```
Xt := Coord[Tree^.Data].X;
Yt := Coord[Tree^.Data].Y;

(* Coordinates of Sibling node of the given node *)

Xs := Coord[Tree^.Sibling^.Data].X;
Ys := Coord[Tree^.Sibling^.Data].Y;
Adjacent := false;
case Find(Xr,Yr) of

(* Situations where the root node can reside *)

 1 : begin

    (* situations where Sibling node can reside *)

     case Find(Xt,Yt) of
      2,3 : if ( Xs = Xmax+Xr ) and (Ys=Yr)
            then Adjacent := true;
      4   : if (Xs = Xr) and (Ys = Yr+1)
            then Adjacent := true;
      5,8 :;
     end; (* CASE *)
    end; (* CASE 1 *)
 2 : begin
     case Find(Xt,Yt) of
      2,3 : if (Xs=Xmax+Xr) and (Ys=Yr)
            then Adjacent:=true;
      4   : begin
            if (Xt-Xmax) = Xr then begin
              if (Xs=Xr) and (Ys=Yr+1)
                then Adjacent:=true;
            end
            else if (Xs=Xr-1) and ( Ys=Yr)
                then Adjacent:=true;
          end; (* CASE 4 *)
      6,9 : if ( Xs=Xmax+Xr-1) and (Ys=Yr)
              then Adjacent := true;
      1   :;
     end; (* CASE *)
    end; (* CASE 2 *)
 3 : begin
     case Find(Xt,Yt) of
      7,10 : if (Xs=Xmax+Xr-1) and (Ys=Yr)
             then Adjacent:=true;
      4    : if (Xs=Xr-1) and (Ys=Yr)
             then Adjacent:=true;
      1,2  :;
     end; (* CASE *)
    end; (* CASE 3 *)
 4 : begin
     if ( Yr = Yt ) then begin
      if ( Xr - Xmax) = Xt then begin
        if (Xs=Xt+1) and (Ys=Yt)
          then Adjacent := true;
      end
```

```
        else if (Xs=Xt) and (Ys=Yt-1)
          then Adjacent := true;
      end
      else begin
        if (Xr-Xmax)=Xt then begin
          if (Xs=Xt) and (Ys=Yt-1)
             then Adjacent := true;
        end
        else if (Xs=Xt-1) and ( Ys=Yt)
          then Adjacent := true;
      end; (* IF ELSE *)
    end ; (* CASE 4 *)
5 : begin
      case Find(Xt,Yt) of
        1,5 : if (Xs=Xmax+Xr) and (Ys=Yt)
               then Adjacent := true;
        4   : begin
               if Yt < Yr then begin
                 if (Xs=Xr+1) and (Ys=Yr)
                    then Adjacent := true;
                end
                else if (Xs=Xr) and (Ys=Yr+1)
                  then Adjacent := true;
              end;
         6,7 : if ( Xs=Xmax-Xr) and (Ys=Yr)
                then Adjacent := true;
         8   :;
      end; (* CADE  *)
    end ; (* CASE 5 *)
6 : begin
      case Find(Xt,Yt) of
        5 : if (Xs=Xmax-Xt) and (Ys=Yt-1)
             then Adjacent := true;
        7 : if (Xs=Xmax-Xr) and (Ys=Yt)
             then Adjacent := true;
        4 : begin
             if Yt < Yr then begin
              if (Xt -Xmax) < Xr then begin
                if (Xs=Xr)and (Ys=Yr-1)
                  then Adjacent := true;
              end
              else if (Xs=Xr+1) and (Ys=Yr)
                  then Adjacent := true;
             end
             else  begin
               if (Xt-Xmax) < Xr then begin
                 if (Xs=Xr-1) and (Ys=Yr)
                   then Adjacent := true
               end
               else if (Xs=Xr) and ( Ys=Yr-1)
                 then Adjacent := true;
             end ; (* IF FLSE *)
           end; (* CASE *)
        2 : if (Xs=Xmax-Xt) and (Ys=Yt)
               then Adjacent := true;
        9 : if ( Xs=Xmax-Xt-1) and (Ys=Yr)
```

```
        then Adjacent := true;
    6 : begin
        if ( Xt > Xr ) and ( Yt = Yr ) then begin
          if ( Xs = Xmax+Xr ) and ( Ys = Yt )
            then Adjacent := true;
        end
        else if ( Xt = Xr ) and ( Yt > Yr ) then begin
          if ( Xs = Xmax+Xt-1 ) and ( Ys = Yr )
            then Adjacent := true;
        end
        else if ( Xt < Xr ) and ( Yt = Yr ) then begin
          if ( Xs = Xmax+Xt ) and ( Ys = Yt-1 )
            then Adjacent := true
        end
        else if ( Xs = Xmax+Xt ) and ( Ys = Yt )
            then Adjacent := true
      end ; (* CASE *)
    end; (* CASE 6 *)
  end; (* CASE 6 *)
   7 : begin
        case Find(Xt,Yt) of
          7,10 : if ( Xs = Xmax+Xt-1 ) and (Ys=Yr)
                  then Adjacent := true;
          4    : begin
                  if ( Yt =Yr ) then begin
                    if (Xs=Xr-1) and (Ys=Yt)
                      then Adjacent := true;
                  end
                  else if ( Xs=Xr ) and (Ys=Yt)
                    then Adjacent := true;
                end;
          5,6  : if ( Xs = Xmax+Xt) and ( Ys=Yt-1 )
                   then Adjacent := true;
          3    :;
        end; (* CASE *)
      end;(* CASE 7 *)
   8 : begin
        case Find(Xt,Yt) of
          1,5 : if ( Xs = Xmax+Xr ) and ( Ys = Yt )
                 then Adjacent := true;
          4   : if ( Xs = Xr+1 ) and ( Ys = Yr )
                 then Adjacent := true;
          9,10 :;
        end; (* CASE *)
      end;(* CASE 8 *)
   9 : begin
        case Find(Xt,Yt) of
          8,9 : if (Xs=Xmax+Xt) and (Ys=Yt-1)
                 then Adjacent := true;
          4   : begin
                 if ( Xt-Xmax ) < Xr then begin
                   if ( Xs=Xr) and (Ys=Yt)
                     then Adjacent := true;
                 end
                 else if (Xs = Xr+1) and ( Ys=Yr )
                   then Adjacent := true
```

```
            end ; (* case *)
        6,2 : if ( Xs = Xmax+Xt ) and ( Ys = Yt )
              then Adjacent := true;
        10 :;
       end; (* CASE *)
      end;(* CASE 9 *)
   10 : begin
       case Find(Xt,Yt) of
        8,9 : if ( Xs = Xmax+Xt ) and ( Ys = Yr - 1 )
              then Adjacent := true;
        4   : if ( Xs = Xr ) and ( Ys = Yr-1 )
              then Adjacent := true;
        3,7 :;
       end; (* CASE *)
      end;(* CASE 9 *)
  end ; (* CASE *)
 end; (* FUNC *)
(*************************************************************************

  This procedure inserts drawing commands by way of a pre-order traversal of
  the directed tree, placing a setpoint command on each node that is a new
  lowest value for the tree. Procedure also takes care of the split edge
  problem.

*************************************************************************)

procedure PutDrawingCommand;
  var
    Head:HeadersType;
    Smaller : real;

(*************************************************************************

  Traverse the contouring tree in preorder, placing drawing commands
  onto nodes and manipulating the split edge problem.

*************************************************************************)

procedure PreOrderPENtrav(var Tree: PointerType);
  var Temp : PointerType;
  begin
    if Tree <> nil then begin
      if Smaller > Tree^.Dnsty
        then begin

      (* Putting Setpoint to the node having the new lowest density value *)

          Tree^.Draw := 1;

          Smaller := Tree^.Dnsty;
        end
      else if Tree^.Sibling <> nil then begin

        (* If SPLIT EDGE exists *)

        if not Adjacent(Tree)
```

```
        then Tree^.Sibling^.Draw := 1;
      end;
      Temp := Tree^.Child;
      PreOrderPENtrav(Temp);
      Temp := Tree^.Sibling;
      PreOrderPENtrav(Temp);
    end; (* IF *)
  end; (* PROC *)

(****************************************************************************

   "PutDrawingCommand" - continued. -

****************************************  ***********************************)

  begin
    Head:=Headers;
    while Head <> nil do begin
      with Head^.Tree^ do begin
        Smaller := NodeDensity[Coord[Data].X,Coord[Data].Y];
        Draw := 2;
      end; (* WITH *)
      PreOrderPENtrav(Head^.Tree);
      Head:= Head^.Child;
    end (* WHILE *)
  end; (* PROC *)

(* ====>  SECTION 4 : PREORDER TRAVERSE OF TREES  <====  *)



(****************************************************************************

  Output all child nodes of the given father pointer.

****************************************************************************)

procedure WriteDescend(Tree:PointerType);
 var i,x,y,D : integer;
 begin
   x := Coord[Tree^.Data].X;
   y := Coord[Tree^.Data].Y;
   D := Tree^.Draw;
   if x > ( Xmax - Xbase ) then
      write(' FATHER NODE X = ',(x-Xmax-0.5):4:2,' Y = ',(y-0.5):4:2)
   else
      write(' FATHER NODE X = ',x:4:2,' Y = ',y:4:2);
   write(' DENSITY= ',NodeDensity[x-Xbase,y-Ybase]:5:2);
   writeln(' DrawCom. ',D:2);
   writeln;write('Children :');
   Tree := Tree^.Child;
   while Tree <> nil do begin
     x := Coord[Tree^.Data].X;
     y := Coord[Tree^.Data].Y;
     if i = 4 then begin
       if x > (Xmax-Xbase) then
```

```
      writeln('X= ',(x-Xmax+0.5):4:2,' Y = ',(y 0.5):4:2,'| ')
    else writeln('X= ',x:4:2,' Y = ',y:4:2,'. ');
    write(' ':10);
    i:= 1;
  end
  else begin
    if x > (Xmax+Xbase) then
      writeln('X= ',(x-Xmax+0.5):4:2,' Y = ',(y+0.5):4:2,'| ')
    else write('X= ',x:4:2,' Y = ',y:4:2,'| ');
    i := i + 1
  end;
  Tree := Tree^.Sibling;
 end;
 writeln('===');writeln;
end ; (* PROC *)

(******************************************************************************

  Procedure traverses the given contouring tree in pre-order and outputs
  information about all nodes with their child nodes.

******************************************************************************)

procedure PreTrav(Tree : PointerType);
 begin
  if Tree <> nil then begin
    WriteDescend(Tree);
    PreTrav(Tree^.Child);
    PreTrav(Tree^.Sibling);
  end;
 end; (* PROC *)
(******************************************************************************

  Traverse all the trees and output information about the nodes of the
  contouring trees.

******************************************************************************)

procedure WriteTreesInPreorderForm ;
  var i :integer;
      Head:HeadersType;
  begin
    Head := Headers;
    i:=1;
    writeln;writeln('TRAVERSE TREES IN PRE-ORDER':35);writeln;
    while Head <> nil do begin
      writeln;writeln('*******     ':20,i:3,'TH TREE    *******');writeln;
      PreTrav(Head^.Tree);
      Head:=Head^.Child;
      i:=i+1;
    end;
  end; (* PROC *)

(* ====> SECTION 5 : TAKING A SLICE OF CONTOUR AT GIVEN CONTOUR LEVEL <==*)
```

```
(**************************************************************************

  Shift the base of the coordinate to what the user wants it to be.

**************************************************************************)

procedure UseBaseCoord;
  var i: integer;
  begin
   for i:=1 to limit do begin
      Coord[i].X := Coord[i].X + Xbase;
      Coord[i].Y := Coord[i].Y + Ybase;
      FromGridToMatrix[Coord[i].X,Coord[i].Y] := i;
   end;
  end; (* PROC *)

(**************************************************************************

  If the root of the tree belongs to situation 6, then the contour lines
  should be  completed, otherwise open contour lines exist. Function
  returns true for the complete contour lines, false for the open contour
  lines.

**************************************************************************)

function IsCompleteDrawing(Root:PointerType;X,Y,Xl,Yl:real):boolean;
  begin
   IsCompleteDrawing:=false;
   if Find(Coord[Root^.Data].X-Xbase,Coord[Root^.Data].Y-Ybase) = 6
    then if ((X<>Xmax) and (Y<>Ymax)) or
         ((Xl<>Xmax) and (Yl<>Ymax))
          then IsCompleteDrawing:=true;
  end;

(**************************************************************************

  Give the coordinates and drawing commands of the contour lines
  at given contour level.

**************************************************************************)

procedure ResultAtGivenContourLevel;
 var i,L : integer;
    X,Y,Z:real;
    Head:HeadersType;
    Root:PointerType;
    first,unwritten :boolean;
    PreCrd:record
          X,Y,Density:real;
           D :0..2;
         end;
    Crd : record
         X,Y:real;
        end;
   EqList:array[1..20] of record
                        Xr,Xs,Yr,Ys:real;
```

```
                    end;
(*****************************************************************************

  If the equivalued edge at the contour level exists, then function returns
  true, and issues a coordinate and drawing instruction pair for that
  equivalued edge.

*****************************************************************************)

function EquivaluedEdge(Root,Subnode:PointerType):boolean:
var i :integer;
   found:boolean;
  begin
    if ( Root^.Dnsty = Subnode^.Dnsty ) and
       ( Root^.Dnsty = ContourLevel ) then begin
        found := false;
        i := 1;
        while ( not (found) ) and ( i<=L ) do begin
          with EqList[i] do
            if (((Xr=Root^.Xval) and ( Yr=Root^.Yval)) and
              ((Xs=Subnode^.Xval) and (Ys=Subnode^.Yval))) or
              (((Xs=Root^.Xval) and ( Ys=Root^.Yval)) and
              ((Xr=Subnode^.Xval) and (Yr=Subnode^.Yval)))
              then  found .= true;
          i:=i+1;
        end; (* WHILE *)
        if not found then begin
          L:=L+1;
          with EqList[L] do begin
            Xr:=Root^.Xval;
            Yr:=Root^.Yval;
            Xs:=Subnode^.Xval;
            Ys:=Subnode^.Yval;
          end; (* WITH *)
          with Root^ do begin
            if first then begin
              X:=Xval;
              Y:=Yval;
              first:=false;
            end;
            writeln(Xval:8:4,Yval:8:4,Z:8:4,'1':8);
          end;
          with Subnode^ do begin
            writeln(Xval:8:4,Yval.8:4,Z:8.4,'0':8);
            PreCrd.X:=Xval
            PreCrd.Y:= Yval;
            PreCrd.D:=0;
            PreCrd.Density:=Dnsty;
          end; (* WITH *)
        end; (* IF *)
        EquivaluedEdge:=true
    end
    else  EquivaluedEdge:=false;
  end; (* FUNC *)
```

```
(****************************************************************************

  This function is used to eliminate duplicate coordinate and drawing commands.

****************************************************************************)

function PreviousCrdCont(X,Y,DENSITY:real;Draw:integer):boolean;
var found:boolean;
   i:integer;
begin
 found:=false;
 PreviousCrdCont:= true;
 if ( PreCrd.Density = DENSITY )
     and ( not ( (( X=PreCrd.X) and (Y=PreCrd.Y))
     and (Draw = PreCrd.D))) then begin
        i := 1,
        while ( not (found) ) and ( i<=L ) do begin
           with EqList[i] do
              if (((Xr=PreCrd.X) and (Yr=PreCrd.Y)) and
                 ((Xs=X) and (Ys=Y))) or
                 (((Xs=PreCrd.X) and (Ys=PreCrd.Y)) and
                 ((Xr=X) and (Yr=Y)))
                 then  found := true;
           i:=i+1;
        end; (* WHILE *)
 end
 else if ( (( X=PreCrd.X) and (Y=PreCrd.Y))
         and (Draw = PreCrd.D))
         then found:=true;
 if not found then begin
    PreCrd.X:=X;
    PreCrd.Y:=Y;
    PreCrd.D:=Draw;
    PreCrd.Density:=DENSITY;
    PreviousCrdCont:= false;
 end;
end;

(****************************************************************************

  Output the coordinates of contour lines at given contour level.

****************************************************************************)

procedure CoordAtGivenLevel(Tree:PointerType);
 var X1,X2,Y1,Y2,Ratio:real;

  begin
   if Tree <> nil then begin
        Root := Tree;
        while ( Root = Root^.pred^.Sibling ) do
           Root := Root^.pred;
        Root := Root^.pred;
        if EquivaluedEdge(Root,Tree) then begin
           CoordAtGivenLevel(Tree^.Child);
           CoordAtGivenLevel(Tree^.Sibling);
```

```
end
else begin
 if Root^.Dnsty > ContourLevel then begin
   if (Tree^.Dnsty =ContourLevel) then begin
     with Tree^ do begin
      if Child <> nil then begin
        if EquivaluedEdge(Tree,Child) then begin
          CoordAtGivenLevel(Child^.Child);
          CoordAtGivenLevel(Child^.Sibling);
        end
        else if not PreviousCrdCont(Xval,Yval,Dnsty,Draw) then
          begin
            if first then begin
              X:=Xval;
              Y:=Yval;
              first:=false;
            end;
            writeln(Xval:8:4,Yval:8:4,Z:8:4,Draw:8);
          end;
      end
      else if not PreviousCrdCont(Xval,Yval,Dnsty,Draw) then
        begin
          if first then begin
            X:=Xval;
            Y:=Yval;
            first:=false;
          end;
          writeln(Xval:8:4,Yval:8:4,Z:8:4,Draw:8);
        end;
     end; (* WITH *)
   end
   else if (Tree^.Dnsty < ContourLevel) then begin

     (* LINEAR INTERPOLLATION *)

     Ratio:=(Root^.Dnsty-ContourLevel)/(Root^.Dnsty-Tree^.Dnsty);
     X1 := Root^.Xval;
     Y1 := Root^.Yval;
     X2 := Tree^.Xval;
     Y2 := Tree^.Yval;
     if (X1-X2) > 0
       then X1:=X2+(X1-X2)*(1-Ratio)
     else X1:=X1+(X2-X1)*Ratio;
     if (Y1-Y2) > 0
       then Y1:=Y2+(Y1-Y2)*(1-Ratio)
     else Y1:=Y1+(Y2-Y1)*Ratio;
     if first then begin
       X:=X1;
       Y:=Y1;
       first:=false;
     end;
   (* Elimination of consequence "setpoint" *)
     if Tree^.Draw = 1 then begin
       Crd.X := X1;
       Crd.Y := Y1;
       unwritten := true;
```

```
            end
            else begin
              if unwritten then writeln(Crd.X:8:4,Crd.Y:8:4,Z:8:4,'1':8);
              if not PreviousCrdCont(X1,Y1,Tree^.Dnsty,Tree^.Draw)
                then  writeln(X1:8:4,Y1:8:4,Z:8:4,Tree^.Draw:8);
              unwritten := false;
            end;
          end
          else  CoordAtGivenLevel(Tree^.Child);
          CoordAtGivenLevel(Tree^.Sibling);
        end;(*IF*)
      end;(* ELSE *)
  end; (* IF *)
end; (* PROC *)

(*****************************************************************************

  Procedure "ResultAtGivenContourLevel".-continued-

 *****************************************************************************)

 begin
   Z:=0;
   while not eof(InpFile) do begin
     readln(InpFile,ContourLevel);
     i:=1;
     Head:=Headers;
     while Head <> nil do begin
       writeln;
       writeln(i:4,'th',' Tree rooted at value ',Head^.Tree^.Dnsty:6:2);
       writeln;
       writeln('Level':9,ContourLevel:5);writeln;
       writeln('X':6,'Y':8,'Z':8,'D':10);writeln;
       first:=true;
       unwritten := false;
       CoordAtGivenLevel(Head^.Tree^.Child);
       if unwritten then writeln(Crd.X:8:4,Crd.Y:8:4,Z:8:4,'1':8);
       if IsCompleteDrawing(Head^.Tree,X,Y,PreCrd.X,PreCrd.Y)
         then if (PreCrd.X <> X) or (PreCrd.Y <> Y)
              then writeln(X:8:4,Y:8:4,Z:8:4,'0':8);
       Head:=Head^.Child;
       i:=i+1;
     end;(* WHILE *)
   end; (* WHILE *)
   writeln;
   write('Column D is the drawing command, i.e. 1 = SETPOINT, 0 = DRAWTO.');
 end;(* PROC *)

(*****************************************************************************
          M A I N   P R O G R A M

 *****************************************************************************)

begin
  Initialize;
  ReadData;
```

```
  WriteInpData;
  CreateIndMatrix;
  WriteIndMatrix;
  CreateTree;
  PutDrawingCommand;
  UseBaseCoord;
  ResultAtGivenContourLevel;
  WriteTreesInPreorderForm;
end.
```

APPENDIX B

PROGRAM OUTPUT FOR THE 2X2 SUBGRID

************* DENSITY VALUES OF NODES IN GRID **************

```
   X -->            1       2       3

Y --> 1- 150.000  40.000  70.000

      2-  30.000  60.000   0.000
```

Locations Of Average Density Values Start After X = 2

INDEGREE-MATRIX

```
1) 0 -1  0 -1 -1
2) 0  3  0  0  0
3) 0 -1  1 -1  0
4) 0  0  0  3  0
5) 0 -1 -1 -1  1
```

1th Tree rooted at value 150.00

Level 50

X	Y	Z	D
2.9091	2.0000	0.0000	1
2.8333	2.1667	0.0000	0
3.0000	2.5000	0.0000	0
2.6667	3.0000	0.0000	1
2.2500	2.7500	0 0000	0
2.0000	2.8333	0.0000	0

1th Tree rooted at value 150.00

Level 100

X	Y	Z	D
2.4545	2.0000	0 0000	1
2.3125	2.3125	0.0000	0
2.0000	2.4167	0 0000	0

Column D is the drawing command, i.e. 1 = SETPOINT, 0 = DRAWTO.

TRAVERSE TREES IN PRE-ORDER

******* 1TH TREE *******

FATHER NODE X = 2.00 Y = 2.00 DENSITY= 150.00 DrawCom: 2

Children :X= 3.00 Y = 2.00 | X= 2.50 Y = 2.50 |
X= 2.00 Y = 3.00 | ===

FATHER NODE X = 3.00 Y = 2.00 DENSITY= 40.00 DrawCom: 1

Children :===

FATHER NODE X = 2.50 Y = 2.50 DENSITY= 70.00 DrawCom: 0

Children :X= 3.00 Y = 2.00 | X= 3.00 Y = 3.00 | X= 2.00 Y = 3.00 | ===

FATHER NODE X = 3.00 Y = 2.00 DENSITY= 40.00 DrawCom: 0

Children :===

FATHER NODE X = 3.00 Y = 3.00 DENSITY= 60.00 DrawCom: 0

Children :X= 3.00 Y = 2.00 | X= 2.00 Y = 3.00 | ===

FATHER NODE X = 3.00 Y = 2.00 DENSITY= 40.00 DrawCom: 0

Children :===

FATHER NODE X = 2.00 Y = 3.00 DENSITY= 30.00 DrawCom: 1

Children :===

FATHER NODE X = 2.00 Y = 3.00 DENSITY= 30.00 DrawCom: 0

Children :===

FATHER NODE X = 2.00 Y = 3.00 DENSITY= 30.00 DrawCom: 0

Children :===

APPENDIX C

PROGRAM OUTPUT FOR THE 3X3 GRID

************* DENSITY VALUES OF NODES IN GRID **************

X -->	1	2	3	4	5
Y --> 1-	150.000	40.000	30.000	70.000	37.500
2-	30.000	60.000	20.000	82.500	35.000
3-	190.000	50.000	10.000	0.000	0.000

Locations Of Average Density Values Start After X = 3

INDEGREE-MATRIX

```
1)  0 -1  0 -1 -1  0  0  0  0  0  0  0  0
2)  0  3  0  0  0 -1  0 -1  0  0  0  0  0
3)  0 -1  2 -1  0  0 -1 -1 -1  0  0  0 -1
4)  0  0  0  5  0  0  0  0  0  0  0  0  0
5)  0 -1 -1 -1  1  0  0  0  0  0  0  0  0
6)  0  0  0  0  0  2 -1  0  0  0  0  0  0
7)  0  0  0  0  0  0  4  0  0  0  0 -1  0
8)  0  0  0  0  0 -1 -1  2  0  0  0  0  0
9)  0  0  0  0  0  0  0  0  3  0  0 -1 -1
10) 0  0  0 -1  0  0  0  0 -1  0 -1  0  0
11) 0  0 -1 -1  0  0  0  0 -1  0  1  0  0
12) 0  0  0  0  0  0  0  0  0  0  0  3  0
13) 0  0  0  0  0  0 -1  0  0  0  0 -1  2
```

1th Tree rooted at value 150.00

Level 50

X	Y	Z	D
2.9091	2.0000	0.0000	1
2.8333	2.1667	0.0000	0
3.0000	2.5000	0.0000	0
3.2222	2.7778	0.0000	0
3.2500	3.0000	0.0000	0
3.2000	3.2000	0.0000	0
3.0000	4.0000	0.0000	0
2.6667	3.0000	0.0000	1
2.2500	2.7500	0.0000	0
2.0000	2.8333	0.0000	0

2th Tree rooted at value 190.00

Level 50

X	Y	Z	D
2.0000	3.1250	0.0000	1
2.1905	3.1905	0.0000	0
2.6667	3.0000	0.0000	0
3.0000	4.0000	0.0000	1
3.0000	4.0000	0.0000	0

1th Tree rooted at value 150.00

Level 100

X	Y	Z	D
2.4545	2.0000	0.0000	1
2.3125	2.3125	0.0000	0
2.0000	2.4167	0.0000	0

2th Tree rooted at value 190.00

Level 100

X	Y	Z	D
2.0000	3.4375	0.0000	1
2.4186	3.5814	0.0000	0
2.6429	4.0000	0.0000	0

Column D is the drawing command, i.e. 1 = SETPOINT. 0 = DRAWTO.

TRAVERSE IN SECOND FORM

******* 1TH TREE *******

FATHER NODE X = 2.00 Y = 2.00 DENSITY= 150.00 DrawCom: 2

Children :X= 3.00 Y = 2.00 | X= 2.50 Y = 2.50
X= 2.00 Y = 3.00 | ===

FATHER NODE X = 3.00 Y = 2.00 DENSITY= 40.00 DrawCom: 1

Children :X= 4.00 Y = 2.00 | X= 3.50 Y = 2.50
===

FATHER NODE X = 4.00 Y = 2.00 DENSITY= 30.00 DrawCom 1

Children :X= 4.00 Y = 3.00 | ===

FATHER NODE X = 4.00 Y = 3.00 DENSITY= 20.00 DrawCom. 1

Children :X= 4.00 Y = 4.00 | ===

FATHER NODE X = 4.00 Y = 4.00 DENSITY= 10.00 DrawCom 1

Children :===

FATHER NODE X = 3.50 Y = 2.50 DENSITY= 37.50 DrawCom: 0

Children :X= 4.00 Y = 2.00 | X= 4.00 Y = 3.00 | ===

FATHER NODE X = 4.00 Y = 2.00 DENSITY= 30.00 DrawCom: 0

Children :X= 4.00 Y = 3.00 | ===

FATHER NODE X = 4.00 Y = 3.00 DENSITY= 20.00 DrawCom: 0

Children :===

FATHER NODE X = 4.00 Y = 3.00 DENSITY= 20.00 DrawCom: 0

Children :===

FATHER NODE X = 2.50 Y = 2.50 DENSITY= 70.00 DrawCom: 0

Children :X= 3.00 Y = 2.00 | X= 3.00 Y = 3.00 | X= 2.00 Y = 3.00 | ===

FATHER NODE X = 3.00 Y = 2.00 DENSITY= 40.00 DrawCom: 0

Children :===

FATHER NODE X = 3.00 Y = 3.00 DENSITY= 60.00 DrawCom: 0

Children :X= 3.00 Y = 2.00 | X= 3.50 Y = 2.50 |
X= 4.00 Y = 3.00 | X= 3.50 Y = 3.50 |
X= 3.00 Y = 4.00 |
X= 2.00 Y = 3.00 | ===

FATHER NODE X = 3.00 Y = 2.00 DENSITY= 40.00 DrawCom: 0

Children :X= 3.50 Y = 2.50 |
===

FATHER NODE X = 3.50 Y = 2.50 DENSITY= 37.50 DrawCom: 0

Children :===

FATHER NODE X = 3.50 Y = 2.50 DENSITY= 37.50 DrawCom: 0

Children :X= 4.00 Y = 3.00 | ===

FATHER NODE X = 4.00 Y = 3.00 DENSITY= 20.00 DrawCom: 0

Children :===

FATHER NODE X = 4.00 Y = 3.00 DENSITY= 20.00 DrawCom: 0

Children :===

FATHER NODE X = 3.50 Y = 3.50 DENSITY= 35.00 DrawCom: 0

Children :X= 4.00 Y = 3.00 | X= 4.00 Y = 4.00 | ===

FATHER NODE X = 4.00 Y = 3.00 DENSITY= 20.00 DrawCom: 0

Children :X= 4.00 Y = 4.00 | ===

FATHER NODE X = 4.00 Y = 4.00 DENSITY= 10.00 DrawCom: 0

Children :===

FATHER NODE X = 4.00 Y = 4.00 DENSITY= 10.00 DrawCom: 0

Children :===

FATHER NODE X = 3.00 Y = 4.00 DENSITY= 50.00 DrawCom: 0

Children :X= 3.50 Y = 3.50 |
X= 4.00 Y = 4.00 | ===

FATHER NODE X = 3.50 Y = 3.50 DENSITY= 35.00 DrawCom: 0

Children :X= 4.00 Y = 4.00 | ===

FATHER NODE X = 4.00 Y = 4.00 DENSITY= 10.00 DrawCom: 0

Children :===

FATHER NODE X = 4.00 Y = 4.00 DENSITY= 10.00 DrawCom: 0

Children :===

FATHER NODE X = 2.00 Y = 3.00 DENSITY= 30.00 DrawCom: 1

Children :===

FATHER NODE X = 2.00 Y = 3.00 DENSITY= 30.00 DrawCom: 0

Children :===

FATHER NODE X = 2.00 Y = 3.00 DENSITY= 30.00 DrawCom: 0

Children :===

******* 2TH TREE *******

FATHER NODE X = 2.00 Y = 4.00 DENSITY= 190.00 DrawCom: 2

Children :X= 2.00 Y = 3.00 | X= 2.50 Y = 3.50 |
X= 3.00 Y = 4.00 | ===

FATHER NODE X = 2.00 Y = 3.00 DENSITY= 30.00 DrawCom: 1

Children :===

FATHER NODE X = 2.50 Y = 3.50 DENSITY= 82.50 DrawCom: 0

Children :X= 2.00 Y = 3.00 | X= 3.00 Y = 3.00 | X= 3.00 Y = 4.00 | ===

FATHER NODE X = 2.00 Y = 3.00 DENSITY= 30.00 DrawCom: 0

Children :===

FATHER NODE X = 3.00 Y = 3.00 DENSITY= 60.00 DrawCom: 0

Children :X= 2.00 Y = 3.00 | X= 3.00 Y = 4.00 | ===

FATHER NODE X = 2.00 Y = 3.00 DENSITY= 30.00 DrawCom: 0

Children :===

FATHER NODE X = 3.00 Y = 4.00 DENSITY= 50.00 DrawCom: 1

Children :===

FATHER NODE X = 3.00 Y = 4.00 DENSITY= 50.00 DrawCom: 0

Children :===

FATHER NODE X = 3.00 Y = 4.00 DENSITY= 50.00 DrawCom: 0

Children :===

APPENDIX D

PROGRAM OUTPUT FOR THE 4X5 GRID

************* DENSITY VALUES OF NODES IN GRID *************

X -->	1	2	3	4	5	6	7
Y --> 1-	20.000	50.000	90.000	70.000	67.500	82.500	57.500
2-	50.000	150.000	40.000	30.000	57.500	70.000	37.500
3-	0.000	30.000	60.000	20.000	57.500	82.500	35.000
4-	10.000	190.000	50.000	10.000	77.500	72.500	17.500
5-	70.000	40.000	10.000	0.000	0.000	0.000	0.000

Locations Of Average Density Values Start After X = 4

INDEGREE-MATRIX

1)	3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
2)	-1	4	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
3)	0	-1	0	-1	-1	0	-1	-1	0	0	0	-1	0	-1	0	-1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
4)	-1	0	0	3	0	0	0	0	0	0	0	0	-1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
5)	-1	-1	0	-1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
6)	0	-1	0	0	0	0	-1	-1	-1	0	-1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
7)	0	0	0	0	0	0	6	0	0	-1	0	0	0	0	0	0	0	-1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
8)	0	-1	0	0	0	0	-1	2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
9)	0	0	0	[illegible]	0	0	0	0	1	-1	-1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
10)	0	0	0	[illegible]	0	0	0	0	0	4	0	0	0	0	0	0	-1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
11)	0	0	0	0	0	0	-1	0	0	-1	2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
12)	0	0	0	0	0	0	0	0	0	0	0	7	-1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
13)	0	0	0	0	0	0	0	0	0	0	0	0	5	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
14)	0	0	0	-1	0	0	0	0	0	0	0	-1	-1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
15)	0	0	0	0	0	0	-1	0	0	0	0	-1	0	0	2	0	-1	-1	0	0	0	-1	0	0	-1	0	0	0	0	0	0	0
16)	0	0	0	0	0	0	-1	0	0	0	0	-1	0	0	-1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
17)	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	4	0	0	0	0	0	0	-1	0	0	0	0	0	0	0	0
18)	0	0	0	0	0	0	0	0	0	-1	0	0	0	0	0	0	-1	2	0	0	0	0	0	0	0	0	0	0	0	0	0	0
19)	0	0	0	0	0	0	0	0	0	0	0	-1	0	0	0	0	0	0	0	-1	-1	-1	-1	0	0	-1	0	-1	0	-1	0	0
20)	0	0	0	0	0	0	0	0	0	0	0	0	-1	0	0	0	0	0	0	4	0	0	0	0	0	0	0	0	0	0	0	0
21)	0	0	0	0	0	0	0	0	0	0	0	-1	-1	0	0	0	0	0	0	-1	1	0	0	0	0	0	0	0	0	0	0	0
22)	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	4	0	-1	-1	0	0	0	-1	0	0	-1
23)	0	0	0	0	0	0	0	0	0	0	0	-1	0	0	-1	0	0	0	0	0	0	-1	1	0	0	0	0	0	0	0	0	0
24)	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	4	0	0	0	0	0	0	-1	0
25)	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	-1	0	0	0	0	0	0	-1	2	0	0	0	0	0	0	0
26)	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	4	0	0	-1	0	0	0
27)	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	-1	0	0	0	0	0	-1	1	0	0	0	0	0
28)	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	-1	0	0	0	0	0	-1	-1	1	0	0	0	0
29)	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	4	0	-1	0
30)	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	-1	0	0	0	-1	0	0	-1	1	0	0

31) 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 3 0
32) 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 -1 0 0 0 0 -1 0 -1 1

1th Tree rooted at value 150.00

Level 50

X	Y	Z	D
2.9091	2.0000	0.0000	1
2.8333	2.1667	0.0000	0
3.0000	2.5000	0.0000	0
3.2222	2.7778	0.0000	0
3.2500	3.0000	0.0000	0
3.2000	3.2000	0.0000	0
3.0000	4.0000	0.0000	0
2.6667	3.0000	0.0000	1
2.2500	2.7500	0.0000	0
2.0000	2.8333	0.0000	0
1.6364	2.6364	0.0000	0
1.4348	2.5652	0.0000	0
1.0000	2.0000	0.0000	0
1.3158	1.3158	0.0000	0
2.0000	1.0000	0.0000	0
2.8824	1.8824	0.0000	1
2.9091	2.0000	0.0000	0

2th Tree rooted at value 90.00

Level 50

X	Y	Z	D
4.0000	1.5000	0.0000	1
3.6364	1.6364	0.0000	0
3.2857	1.7143	0.0000	0
3.0000	1.8000	0.0000	0
2.8824	1.8824	0.0000	0
2.0000	1.0000	0.0000	1
2.0000	1.0000	0.0000	0

3th Tree rooted at value 190.00

Level 50

X	Y	Z	D
3.0000	4.0000	0.0000	1
3.0000	4.0000	0.0000	0
2.6800	4.6800	0.0000	0
2.1538	4.8462	0.0000	0
2.0000	4.9333	0.0000	0
1.8667	4.8667	0.0000	0
1.6667	5.0000	0.0000	0
1.0000	4.6667	0.0000	1

1.2963	4.2963	0.0000	0
1.2222	4.0000	0.0000	0
1.4211	3.5789	0.0000	0
1.4348	3.4348	0.0000	0
1.6364	3.3636	0.0000	0
2.0000	3.1250	0.0000	0
2.1905	3.1905	0.0000	0
2.6667	3.0000	0.0000	0
3.0000	4.0000	0.0000	1
3.0000	4.0000	0.0000	0

1th Tree rooted at value 150.00

Level 100

X	Y	Z	D
2.4545	2.0000	0.0000	1
2.3125	2.3125	0.0000	0
2.0000	2.4167	0.0000	0
1.7297	2.2703	0.0000	0
1.5000	2.0000	0.0000	0
1.6970	1.6970	0.0000	0
2.0000	1.5000	0.0000	0
2.3704	1.6296	0.0000	0
2.4545	2.0000	0.0000	0

2th Tree rooted at value 90.00

Level 100

X	Y	Z	D

3th Tree rooted at value 190.00

Level 100

X	Y	Z	D
2.6429	4.0000	0.0000	1
2.3830	4.3830	0.0000	0
2.0000	4.6000	0.0000	0
1.6000	4.4000	0.0000	0
1.5000	4.0000	0.0000	0
1.6604	3.6604	0.0000	0
2.0000	3.4375	0.0000	0
2.4186	3.5814	0.0000	0
2.6429	4.0000	0.0000	0

Column D is the drawing command, i.e. 1 = SETPOINT, 0 = DRAWTO

TRAVERSE TREES IN PRE-ORDER

******* 1TH TREE *******

FATHER NODE X = 2.00 Y = 2.00 DENSITY= 150.00 DrawCom: 2

Children :X= 3.00 Y = 2.00 | X= 2.50 Y = 2.50 |
X= 2.00 Y = 3.00 | X= 1.50 Y = 2.50 |
X= 1.00 Y = 2.00 |
X= 1.50 Y = 1.50 |
X= 2.00 Y = 1.00 | X= 2.50 Y = 1.50
===

FATHER NODE X = 3.00 Y = 2.00 DENSITY= 40.00 DrawCom: 1

Children :X= 4.00 Y = 2.00 | X= 3.50 Y = 2.50
===

FATHER NODE X = 4.00 Y = 2.00 DENSITY= 30.00 DrawCom: 1

Children :X= 4.00 Y = 3.00 | ===

FATHER NODE X = 4.00 Y = 3.00 DENSITY= 20.00 DrawCom: 1

Children :X= 4.00 Y = 4.00 | ===

FATHER NODE X = 4.00 Y = 4.00 DENSITY= 10.00 DrawCom: 1

Children :X= 4.00 Y = 5.00 | ===

FATHER NODE X = 4.00 Y = 5.00 DENSITY= 0.00 DrawCom: 1

Children :===

FATHER NODE X = 3.50 Y = 2.50 DENSITY= 37.50 DrawCom: 0

Children :X= 4.00 Y = 2.00 X= 4.00 Y = 3.00 ===

FATHER NODE X = 4.00 Y = 2.00 DENSITY= 30.00 DrawCom: 0

Children :X= 4.00 Y = 3.00 ===

FATHER NODE X = 4.00 Y = 3.00 DENSITY= 20.00 DrawCom: 0

Children :===

FATHER NODE X = 4.00 Y = 3.00 DENSITY= 20.00 DrawCom: 0

Children :===

FATHER NODE X = 2.50 Y = 2.50 DENSITY= 70.00 DrawCom: 0

Children :X= 3.00 Y = 2.00 X= 3.00 Y = 3.00 X= 2.00 Y = 3.00 ===

FATHER NODE X = 3.00 Y = 2.00 DENSITY= 40.00 DrawCom: 0

Children :===

FATHER NODE X = 3.00 Y = 3.00 DENSITY= 60.00 DrawCom: 0

Children :X= 3.00 Y = 2.00 | X= 3.50 Y = 2.50 |
X= 4.00 Y = 3.00 | X= 3.50 Y = 3.50 |
X= 3.00 Y = 4.00 |
X= 2.00 Y = 3.00 | ===

FATHER NODE X = 3.00 Y = 2.00 DENSITY= 40.00 DrawCom: 0

Children :X= 3.50 Y = 2.50 |
===

FATHER NODE X = 3.50 Y = 2.50 DENSITY= 37.50 DrawCom: 0

Children :===

FATHER NODE X = 3.50 Y = 2.50 DENSITY= 37.50 DrawCom: 0

Children :X= 4.00 Y = 3.00 | ===

FATHER NODE X = 4.00 Y = 3.00 DENSITY= 20.00 DrawCom: 0

Children :===

FATHER NODE X = 4.00 Y = 3.00 DENSITY= 20.00 DrawCom: 0

Children :===

FATHER NODE X = 3.50 Y = 3.50 DENSITY= 35.00 DrawCom: 0

Children :X= 4.00 Y = 3.00 | X= 4.00 Y = 4.00 | ===

FATHER NODE X = 4.00 Y = 3.00 DENSITY= 20.00 DrawCom: 0

Children :X= 4.00 Y = 4.00 | ===

FATHER NODE X = 4.00 Y = 4.00 DENSITY= 10.00 DrawCom: 0

Children :===

FATHER NODE X = 4.00 Y = 4.00 DENSITY= 10.00 DrawCom: 0

Children :===

FATHER NODE X = 3.00 Y = 4.00 DENSITY= 50.00 DrawCom: 0

Children :X= 3.50 Y = 3.50 |
X= 4.00 Y = 4.00 | X= 3.50 Y = 4.50 |
X= 3.00 Y = 5.00 | ===

FATHER NODE X = 3.50 Y = 3.50 DENSITY= 35.00 DrawCom: 0

Children :X= 4.00 Y = 4.00 | ===

FATHER NODE X = 4.00 Y = 4.00 DENSITY= 10.00 DrawCom: 0

Children :===

FATHER NODE X = 4.00 Y = 4.00 DENSITY= 10.00 DrawCom: 0

Children :===

FATHER NODE X = 3.50 Y = 4.50 DENSITY= 17.50 DrawCom: 0

Children :X= 4.00 Y = 4.00 | X= 4.00 Y = 5.00 | X= 3.00 Y = 5.00 | ===

FATHER NODE X = 4.00 Y = 4.00 DENSITY= 10.00 DrawCom: 0

Children :X= 4.00 Y = 5.00 | ===

FATHER NODE X = 4.00 Y = 5.00 DENSITY= 0.00 DrawCom: 0

Children :===

FATHER NODE X = 4.00 Y = 5.00 DENSITY= 0.00 DrawCom: 0

Children :===

FATHER NODE X = 3.00 Y = 5.00 DENSITY= 10.00 DrawCom: 0

Children :X= 4.00 Y = 5.00 | ===

FATHER NODE X = 4.00 Y = 5.00 DENSITY= 0.00 DrawCom: 0

Children :===

FATHER NODE X = 3.00 Y = 5.00 DENSITY= 10.00 DrawCom: 0

Children :===

FATHER NODE X = 2.00 Y = 3.00 DENSITY= 30.00 DrawCom: 1

Children :X= 1.00 Y = 3.00 | ===

FATHER NODE X = 1.00 Y = 3.00 DENSITY= 0.00 DrawCom: 0

Children :===

FATHER NODE X = 2.00 Y = 3.00 DENSITY= 30.00 DrawCom: 0

Children :===

FATHER NODE X = 2.00 Y = 3.00 DENSITY= 30.00 DrawCom: 0

Children :===

FATHER NODE X = 1.50 Y = 2.50 DENSITY= 57.50 DrawCom: 0

Children :X= 2.00 Y = 3.00 | X= 1.00 Y = 3.00 | X= 1.00 Y = 2.00 | ===

FATHER NODE X = 2.00 Y = 3.00 DENSITY= 30.00 DrawCom: 0

Children :X= 1.00 Y = 3.00 | ===

FATHER NODE X = 1.00 Y = 3.00 DENSITY= 0.00 DrawCom: 0

Children :===

FATHER NODE X = 1.00 Y = 3.00 DENSITY= 0.00 DrawCom: 0

Children :===

FATHER NODE X = 1.00 Y = 2.00 DENSITY= 50.00 DrawCom: 0

Children :X= 1.00 Y = 3.00 | X= 1.00 Y = 1.00 | ===

FATHER NODE X = 1.00 Y = 3.00 DENSITY= 0.00 DrawCom: 0

Children :===

FATHER NODE X = 1.00 Y = 1.00 DENSITY= 20.00 DrawCom: 1

Children :===

FATHER NODE X = 1.00 Y = 2.00 DENSITY= 50.00 DrawCom: 0

Children :===

FATHER NODE X = 1.50 Y = 1.50 DENSITY= 67.50 DrawCom: 0

Children :X= 1.00 Y = 2.00 | X= 1.00 Y = 1.00 | X= 2.00 Y = 1.00 | ===

FATHER NODE X = 1.00 Y = 2.00 DENSITY= 50.00 DrawCom: 0

Children :X= 1.00 Y = 1.00 | ===

FATHER NODE X = 1.00 Y = 1.00 DENSITY= 20.00 DrawCom: 0

Children :===

FATHER NODE X = 1.00 Y = 1.00 DENSITY= 20.00 DrawCom: 0

Children :===

FATHER NODE X = 2.00 Y = 1.00 DENSITY= 50.00 DrawCom: 0

Children :X= 1.00 Y = 1.00 | ===

FATHER NODE X = 1.00 Y = 1.00 DENSITY= 20.00 DrawCom: 0

Children :===

FATHER NODE X = 2.00 Y = 1.00 DENSITY= 50.00 DrawCom: 0

Children :===

FATHER NODE X = 2.50 Y = 1.50 DENSITY= 82.50 DrawCom: 0

Children :X= 2.00 Y = 1.00 | X= 3.00 Y = 2.00 | ===

FATHER NODE X = 2.00 Y = 1.00 DENSITY= 50.00 DrawCom: 0

Children :===

FATHER NODE X = 3.00 Y = 2.00 DENSITY= 40.00 DrawCom: 1

Children :===

******* 2TH TREE *******

FATHER NODE X = 3.00 Y = 1.00 DENSITY= 90.00 DrawCom: 2

Children :X= 4.00 Y = 1.00 | X= 3.50 Y = 1.50 |
X= 3.00 Y = 2.00 | X= 2.50 Y = 1.50 |
X= 2.00 Y = 1.00 |
===

FATHER NODE X = 4.00 Y = 1.00 DENSITY= 70.00 DrawCom: 1

Children :X= 4.00 Y = 2.00 | X= 3.50 Y = 1.50 |
===

FATHER NODE X = 4.00 Y = 2.00 DENSITY= 30.00 DrawCom: 1

Children :===

FATHER NODE X = 3.50 Y = 1.50 DENSITY= 57.50 DrawCom: 0

Children :X= 4.00 Y = 2.00 | X= 3.00 Y = 2.00 | ===

FATHER NODE X = 4.00 Y = 2.00 DENSITY= 30.00 DrawCom: 0

Children :===

FATHER NODE X = 3.00 Y = 2.00 DENSITY= 40.00 DrawCom: 0

Children :X= 4.00 Y = 2.00 | ===

FATHER NODE X = 4.00 Y = 2.00 DENSITY= 30.00 DrawCom: 0

Children :===

FATHER NODE X = 3.50 Y = 1.50 DENSITY= 57.50 DrawCom: 0

Children :X= 3.00 Y = 2.00 ===

FATHER NODE X = 3.00 Y = 2.00 DENSITY= 40.00 DrawCom: 0

Children :===

FATHER NODE X = 3.00 Y = 2.00 DENSITY= 40.00 DrawCom: 0

Children :===

FATHER NODE X = 2.50 Y = 1.50 DENSITY= 82.50 DrawCom: 0

Children :X= 3.00 Y = 2.00 | X= 2.00 Y = 1.00 | ===

FATHER NODE X = 3.00 Y = 2.00 DENSITY= 40.00 DrawCom: 0

Children :===

FATHER NODE X = 2.00 Y = 1.00 DENSITY= 50.00 DrawCom: 1

Children :===

FATHER NODE X = 2.00 Y = 1.00 DENSITY= 50.00 DrawCom: 0

Children :===

******* 3TH TREE *******

FATHER NODE X = 2.00 Y = 4.00 DENSITY= 190.00 DrawCom: 2

Children :X= 3.00 Y = 4.00 | X= 2.50 Y = 4.50 |
X= 2.00 Y = 5.00 | X= 1.50 Y = 4.50 |
X= 1.00 Y = 4.00 |
X= 1.50 Y = 3.50 |
X= 2.00 Y = 3.00 | X= 2.50 Y = 3.50 |
===

FATHER NODE X = 3.00 Y = 4.00 DENSITY= 50.00 DrawCom: 1

Children :===

FATHER NODE X = 2.50 Y = 4.50 DENSITY= 72.50 DrawCom: 0

Children :X= 3.00 Y = 4.00 | X= 3.00 Y = 5.00 | X= 2.00 Y = 5.00 ===

FATHER NODF X = 3.00 Y = 4.00 DENSITY= 50.00 DrawCom: 0

Children :X= 3.00 Y = 5.00 | ===

FATHER NODE X = 3.00 Y = 5.00 DENSITY= 10.00 DrawCom: 1

Children :===

FATHER NODE X = 3.00 Y = 5.00 DENSITY= 10.00 DrawCom: 0

Children :===

FATHER NODE X = 2.00 Y = 5.00 DENSITY= 40.00 DrawCom: 0

Children :X= 3.00 Y = 5.00 | ===

FATHER NODE X = 3.00 Y = 5.00 DENSITY= 10.00 DrawCom: 0

Children :===

FATHER NODE X = 2.00 Y = 5.00 DENSITY = 40.00 DrawCom: 0

Children :===

FATHER NODE X = 1.50 Y = 4.50 DENSITY = 77.50 DrawCom: 0

Children :X= 2.00 Y = 5.00 | X= 1.00 Y = 5.00 | X= 1.00 Y = 4.00 | ===

FATHER NODE X = 2.00 Y = 5.00 DENSITY= 40.00 DrawCom: 0

Children :===

FATHER NODE X = 1.00 Y = 5.00 DENSITY= 70.00 DrawCom: 0

Children :X= 2.00 Y = 5.00 | X= 1.00 Y = 4.00 | ===

FATHER NODE X = 2.00 Y = 5.00 DENSITY= 40.00 DrawCom: 0

Children :===

FATHER NODE X = 1.00 Y = 4.00 DENSITY= 10.00 DrawCom: 1

Children :X= 1.00 Y = 3.00 | ===

FATHER NODE X = 1.00 Y = 3.00 DENSITY= 0.00 DrawCom: 1

Children :===

FATHER NODE X = 1.00 Y = 4.00 DENSITY= 10.00 DrawCom: 0

Children :===

FATHER NODE X = 1.00 Y = 4.00 DENSITY= 10.00 DrawCom: 0

Children :===

FATHER NODE X = 1.50 Y = 3.50 DENSITY= 57.50 DrawCom: 0

Children :X= 1.00 Y = 4.00 | X= 1.00 Y = 3.00 | X= 2.00 Y = 3.00 | ===

FATHER NODE X = 1.00 Y = 4.00 DENSITY= 10.00 DrawCom: 0

Children :X= 1.00 Y = 3.00 | ===

FATHER NODE X = 1.00 Y = 3.00 DENSITY= 0.00 DrawCom: 0

Children :===

FATHER NODE X = 1.00 Y = 3.00 DENSITY= 0.00 DrawCom: 0

Children :===

FATHER NODE X = 2.00 Y = 3.00 DENSITY= 30.00 DrawCom: 0

Children :X= 1.00 Y = 3.00 | ===

FATHER NODE X = 1.00 Y = 3.00 DENSITY= 0 00 DrawCom: 0

Children :===

FATHER NODE X = 2.00 Y = 3.00 DENSITY= 30.00 DrawCom: 0

Children :===

FATHER NODE X = 2.50 Y = 3.50 DENSITY= 82.50 DrawCom: 0

Children :X= 2.00 Y = 3.00 | X= 3.00 Y = 3.00 | X= 3.00 Y = 4.00 | ===

FATHER NODE X = 2.00 Y = 3.00 DENSITY= 30.00 DrawCom: 0

Children :===

FATHER NODE X = 3.00 Y = 3.00 DENSITY= 60.00 DrawCom: 0

Children :X= 2.00 Y = 3.00 | X= 3.00 Y = 4.00 | ===

FATHER NODE X = 2.00 Y = 3.00 DENSITY= 30.00 DrawCom: 0

Children :===

FATHER NODE X = 3.00 Y = 4.00 DENSITY= 50.00 DrawCom: 1

Children :===

FATHER NODE X = 3.00 Y = 4.00 DENSITY= 50.00 DrawCom: 0

Children :===

LIST OF REFERENCES

1. Zyda, Michael J., *A Decomposable Algorithm For Contour Surface Display Generation*, Dept. of Computer Science, Naval Postgraduate School, Monterey, California, August 1984.

2. Zyda, Michael J., *Algorithm Directed Architectures for Real-Time Surface Display Generation*, D.Sc. Dissertation, Dept. of Computer Science, Washington Univ, St.Louis, Missouri, 1984.

3. Zyda, Michael J., *A Contour Display Generation Algorithm For VLSI Implementation, Selected Reprints on VLSI Technologies and Computer Graphics*, Compiled by Henry Fuchs, p. 459, Spring, Maryland: IEEE Computer Society Press, 1983.

4. Zyda, Michael J., *A Contour Display Generation Algorithm for VLSI implementation*. Computer Graphics: A Quarterly Report of SIGGRAPH-ACM, Vol. 16, No. 3 (July 1982), p. 135.

5. Zyda, Michael J., *Multiprocessor Considerations in the Design of a Real-Time Contour Display Generator*, Technical Memorandum 42, St. Louis: Department of Computer Science. Washington University, December 1981.

6. Barry, C.D. and Sucher, J. H., *Interactive Real-Time Contouring of Density Maps*, American Crystallogrphic Association Winter Meeting, Hononlulu, March 1979. Poster Session.

7. Faber, D.H., Rutten-Keulemans. E.W.M., and Altona. C., *Computer Plotting of Contour Maps : An Improved Method*, Computer & Chemistry, Vol. 3, pp. 51-55, Great Britain: Pergamon Press Ltd., 12979.

8. Wright, Thomas and Humbrecht John, *ISOSRF - An Algorithm for Plotting Iso-Valued Surfaces of a Function of Three Variables*, Computer Graphics: A Quarterly Report of SIGGRAPH-ACM, Vol. 13, No. 2 (August 1979), pp. 182-189.

9. Dutton, G., *An Extensible Approach to Imagery of Gridded Data*, Computer Graphics: A quarterly Report of SIGGRAPH-ACM, Vol. 11, No. 2 (July 1977), p.159.

10. Gold, G.M., *Automated Contour Mapping Using Triangular Element Data Structures and An Interpolant Over Each Irregular Triangular Domain,* Computer Graphics: A Quarterly Report of SIGGRAPH-ACM, Vol. 11, No. 2 (July 1977), p. 170.

11. Cottafava, G. and Le Moli, G., *Automatic Contour Map,* Communications of the ACM, Vol 12, No. 7 (July 1969), pp. 386-391.

12. Dayhoff, M.O., *A Contour-Map Program for X-Ray Crystallography,* Communications of ACM, Vol. 6, No. 10 (October 1963), pp. 620-622.

13. McLain, D.H., *Drawing Contours from Arbitrary Data Points,* The Computer Journal, Vol. 17, No. 4 (1974), p. 318.

14. Sabin, M.A., *Contouring -- A Review of Methods for Scattered Data, Mathematical Methods in Computer Graphics and Design,* Edited by K.W. Brodlie, pp. [illegible]3, Great Britain: Academic Press, 1980.

15. Sutcliffe, D.C., *Contouring Over Rectangular and Skewed Rectangular Grids-An introduction,* Mathematical Methods in Computer Graphics and Design. Edited by K.W. Brodlie, pp. 39-62, great Britain: Academic Press, 1980.

16. Sutcliffe, D.C., *A remark on a Contouring Algorithm,* The Computer Journal, Vol. 19. No. 4 (1976a), p. 233.

17. Sutcliffe, D.C., *An Algorithm for Drawing the Curve f(x,y)=0,* The Computer Journal, Vol. 19, No. 3, (1976b), p. 246.

18. Even, *Shimon Graph Algorithms,* Potomac, Maryland: Computer Science Press, 1979.

19. Horowitz, Ellis and Sahni, Sartaj. *Fundamentals of Data Structures.* Potomac, Maryland: Computer Science Press. 1977. Chapter 5.

INITIAL DISTRIBUTION LIST

			No. Copies
1.	Defense Technical Information Center Cameron Station Alexandria, Virginia 22304-6145		2
2.	Superintendent Attn: Library (Code 0142) Naval Postgraduate School Monterey, California 93943-5100		2
3.	Chairman (Code 52) Department of Computer Science Naval Postgraduate School Monterey, California 93943-5100		1
4.	Computer Technology Programs (Code 37) Naval Postgraduate School Monterey, California 93943-5100		1
5.	Michael J. Zyda (Code 52) Department of Computer Science Naval Postgraduate School Monterey, California 93943-5100		5
6.	Daniel L. Davis (Code 52) Department of Computer Science Naval Postgraduate School Monterey, California 93943-5100		1
7	Lt. Mustafa Sahintepe Kaleardi Mah. Kisla Yolu Uzeri No: 23/5 Tokat / TURKEY		10
8.	Patricia Hart C/O John Camanse 239 Waena St Whitmore Oahu HI. 96786		1
9.	Turk Hava Kuvvetleri Komutanligi Per E.t. D. Bsk. Bakanliklar Ankara / TURKEY		2

10. **Hava Harb Okulu Kutuphanesi** 1
Yesilyurt Istanbul / TURKEY

11. **Istanbul Texnik Universitesi** 1
Kutuphane
Gumussuyu Istanbul / TURKEY

12. **Ortadogu Texnik Universitesi** 1
Kutuphane
Ankara/TURKEY

13. **Hava Teknik Okullar Komutanligi** 1
Kutuphane
Gaziemir Izmir / TURKEY

14. **Hava Teknik Okullar Komutanligi** 1
Muhabere Okulu
Kutuphane
Gaziemir Izmir / TURKEY